DBMS in Hindi

BccFalna.com
097994-55505

Kuldeep Chand

This EBook is basically useful if you want to learn to develop **Professional Application Level Database,** so that you can understand the various theoratical processes related to Database Designing like *Requirement Identification, Database Normalization, Entity Relationship **(ER)** Diagram Creation, Data Flow Diagram **(DFD)***, etc…

This eBook is based on various kinds of Database Related Problems and then Identifying its Solutions, so that you can create a **Database Schema** on the basis of the requirement.

Relationship of a Relation Database is the main fundamental concept and in this eBook, I have tried my best to explain Various Relationship Concepts like **One-to-One, One-to-Many** and **Many-to-Many** with easy to understand Examples.

Database Normalization is another the most important concept for developing a well performing Database, so I have included it with easy to understand Detailed Examples too.

DBMS

In Hindi

■■■
Kuldeep Chand

Betalab Computer Center
Falna

DBMS-RDBMS in Hindi

Lead Editors: **Kuldeep Chand**

Distributed to the book trade worldwide by Betalab Computer Center, Behind of Vidhya Jyoti School, Falna Station Dist. Pali (Raj.) Pin 306116

e-mail bccfalna@gmail.com

or

visit http://www.bccfalna.com

For information on translations, please contact BetaLab Computer Center, Behind of Vidhya Jyoti School, Falna Station Dist. Pali (Raj.) Pin 306116

Phone **97994-55505**

This book is dedicated to those

who really wants to be

a

PROFESSIONAL DEVELOPER

INDEX OF CONTENTS

Table of Contents

DBMS

DATABASE MANAGEMEMNT SYSTEM

Database Management System

इससे पहले कि हम Oracle को समझें, हमें सबसे पहले Database के विभिन्न Concepts को बेहतर तरीके से समझना होगा, ताकि हम ये समझ सकें कि ऐसी कौनसी समस्याए हैं, जिनके Solution के रूप में Oracle जैसा DBMS Software Develop किया गया है। तो सबसे पहले हम Relational Database व Relational Database को Design करने के Process के बारे में जानेंगे।

चूंकि Relational Database Designing एक बहुत ही जटिल प्रक्रिया है और हम Designing व Implementation दोनों को दो अलग रूपों में देख सकते हैं। इसलिए इस पुस्तक को भी हमने दो भागों में विभाजित किया है। पहले भाग के अन्तर्गत हम Relational Database Designing से संबंधित विभिन्न बातों को अच्छी तरह से समझने की कोशिश करेंगे, जबकि दूसरे Section में हम ये जानेंगे कि पहले Section के आधार पर Designing किए गए किसी Database को Oracle में के साथ किस तरह से Implement किया जाता है।

चूंकि Designing व Implementation दो एकदम अलग Subjects होते हुए भी एक दूसरे से गहराई के साथ आपस में जुडे हुए हैं, इसलिए इस पुस्तक में कई उदाहरण व समान बातें आपको बार–बार देखने व पढने को मिल सकती हैं। हमने कई उदाहरणों व तथ्यों को बार–बार इसलिए Repeat किया है, ताकि पुस्तक के Contents का Flow बना रहे।

Introduction

सभ्यता की शुरूआत से ही मानव को Information की जरूरत रही है। इसीलिए वह समय–समय पर सूचनाओं को एकत्रित करने व उन सूचनाओं के आधार पर सही व उचित निर्णय लेने के नए व विकसित तरीके खोजता रहा है। सूचना की आवश्यकता व महत्व के कारण सबसे पहला आविष्कार कागज व कलम का हुआ।

जैसे–जैसे मानव का विकास होता गया वैसे–वैसे उसने नए शहर, राज्य व देश बनाए और उन देशों के बीच व्यापार व वाणिज्य के कारण विभिन्न सम्बंध बने और आज केवल व्यापार व वाणिज्य ही नहीं बल्कि जीवन की लगभग हर सूचना का Internet के माध्यम से इन देशों के बीच आदान प्रदान हो रहा है। कृषि क्रांति व औधोगिक क्रांति के बाद आज हम सूचना क्रांति के युग में जी रहे हैं।

पहले सूचनाओं को मिट्टी के बर्तनों पर चित्रात्मक रूप में व शब्दों के रूप में लिखा जाता था। फिर कागज व कलम के विकास से इन पर विभिन्न सूचनाओं को Store करके रखा जाने लगा और आज हम इन्हीं सूचनाओं को Computer पर Manage करते हैं।

विभिन्न प्रकार के आंकडों (**Data**) का संकलन (Collection) करना और फिर उन आंकडों को विभिन्न प्रकार से वर्गीकृत (Classify) करके उनका विश्लेषण (Analyze) करना तथा उचित समय

पर उचित निर्णय लेने की क्षमता प्राप्त करना, इस पूरी प्रक्रिया को Computer की भाषा में **Data Processing** करना कहा जाता है।

आज हम देख सकते हैं कि Computer का उपयोग विभिन्न प्रकार के Business से सम्बंधित सूचनाओं को Store, Manage व Process करने के लिए किया जाता है। Business भले छोटा हो या बडा, Computer की अपनी कुछ विशेषताओं के कारण सभी प्रकार के Businesses में इन्सानों के साथ–साथ अब Computer का अधिकाधिक प्रयोग किया जाने लगा है।

जब हम Computer पर किसी समस्या का समाधान प्राप्त करना चाहते हैं, तब उस समस्या को हमें Computer में एक Software Application के रूप में Represent करना होता है। Computer हमेंशा किसी ना किसी Software Application के आधार पर ही काम करता है और यदि किसी समस्या का समाधान प्राप्त करने के लिए Develop किया गया Software Application पूरी तरह से सही हो, तो Computer कभी भी गलत Result प्रदान नहीं करता है। यही Computer की सबसे पहली व सबसे बडी विशेषता है, कि वह हमेंशा Software Application के आधार पर ही काम करता है, इसलिए उससे प्राप्त होने वाला Result कभी भी गलत नहीं होता। यानी Computer हमेंशा **Accurate** Result Provide करता है।

लेकिन यदि समस्या का समाधान प्राप्त करने के लिए Develop किए गए Software Application के Design में कोई गडबड हो, तो Computer स्वयं उस गलती को पकड कर सही निर्णय लेने में सक्षम नहीं होता है। इस स्थिति में Computer गलत Result भी दे सकता है, जिसके बारे में हमेंशा Software Application Develop करने वाले Programmer को सावधान रहना होता है।

Computer की दूसरी विशेषता ये है कि Computer Calculations से सम्बंधित काम बहुत ही सफलतापूर्वक व तेजी से कर सकता है, जिन्हें करने में इन्सानों को काफी समय लगता है और विशेष सावधानी बरतनी पडती है। Computer Calculation से सम्बंधित गलतियां कभी भी नहीं करता है, जब तक कि Computer जिस Software Application के आधार पर Calculation कर रहा है, उस Application को ही गलत तरह की Calculations को Perform करने के लिए Design ना किया गया हो।

चूंकि Computer Calculations को **Fastly** व **Accurately** Perform करता है, इसलिए Computer के कारण Business Man को Calculation के प्रति विशेष सावधान रहने की जरूरत नहीं रह जाती है। इस स्थिति में एक Business Man अपने व्यापार को बढाने से सम्बंधित निर्णयों को जल्दी से व आसानी से ले सकने में सक्षम हो जाता है।

Computer की एक तीसरी विशेषता ये है कि आम इन्सानों की तरह ही Computer भी सूचनाओं को याद रख सकता है। लेकिन चूंकि इन्सानों की एक कमी ये भी है कि वे यदि किसी Information को लम्बे समय तक उपयोग में ना लें, तो वे बातों को भूल जाते हैं, जबकि Computer पर Stored सूचनाओं को Computer कभी भी नहीं भूलता है।

Computer पर Stored सूचनाओं को एक Business Man सालों बाद भी ज्यों का त्यों प्राप्त कर सकता है, जिससे एक Business Man को इन्सानी गलतियों से होने वाली परेशानियों का भी सामना नहीं करना पडता है। ऐसी ही और भी बहुत सी विशेषताओं के कारण ही आज हर Business Man चाहे वह छोटा Business कर रहा हो या बडा, Computer पर ही अपने Business से सम्बंधित सूचनाओं को Manage करता है।

हर Business Man Computer पर अपने Business से सम्बंधित जरूरी Data को Maintain करता है, ताकि वह अपने Business से सम्बंधित जानकारियों को अच्छी तरह से व तेजी से प्राप्त कर सके व अपने Business से सम्बंधित निर्णय ले सके।

Data – Value or a Set of Values

असिद्ध तथ्य (Facts) अंक (Figures) व सांख्यिकी (Statics) का वह समूह जिस पर प्रक्रिया (Processing) करने पर एक अर्थपूर्ण (Meaningful) सूचना (Information) प्राप्त (Generate) हो, **Data** कहलाता है। Data मान या मानों का एक समूह (Value or a Set of Values) होता है, जिसके आधार पर (After Processing) हम निर्णय (Decision) लेते हैं।

इसे एक उदाहरण द्वारा समझने की कोशिश करते हैं। संख्याएं (0 से 9 तक) कुल दस ही होती हैं। लेकिन यदि इन्हें एक व्यवस्थित क्रम में रख दिया जाए, तो एक सूचना Generate होती है। इसलिए ये संख्याएं Data हैं।

अंग्रेजी भाषा में Small व Capital Letters के कुल 52 Characters ही होते हैं, लेकिन यदि इन्हें एक सुव्यवस्थित क्रम में रखा जाए, तो हजारों पुस्तकें बन सकती हैं। इसलिए ये Characters Data हैं।

Computer में हम इन्हीं दो रूपों में वास्तविक जीवन की विभिन्न बातों को Store करते हैं और उन पर Processing करके आवश्यकतानुसार Information Generate करते हैं। जैसे किसी School के विभिन्न Students की ये जानकारी रखनी हो कि किसी Class में कौन–कौन से Students हैं, उनका Serial Number क्या है और वे किस Address पर रहते हैं, तो ये सभी तथ्य असिद्ध रूप में Computer के लिए **Data** हैं क्योंकि किसी Student के Serial Number को 0 से 9 के कुछ अंकों के समूह के रूप में Express किया जाता है और Student का नाम व पता Characters के एक सुव्यवस्थित समूह के रूप में Express किया जाता है।

जब 0 से 9 तक के कुछ अंकों को एक समूह में व्यवस्थित किया जाता है तब किसी एक Student का एक Serial Number बन जाता है और जब विभिन्न Characters को एक समूह में व्यवस्थित किया जाता है, तब किसी Student का नाम व Address बन जाता है। ये नाम व Address ही किसी Student की कुछ **Information** प्रदान करते हैं।

Processing – Generating Results

Data जैसे कि अक्षर, अंक, सांख्यकी Statics या किसी चित्र को सुव्यवस्थित करना या उनकी Calculation करना, **Processing** कहलाता है। किसी भी Processing मे निम्न काम होते हैं:

Calculation	किसी मान को जोडना, घटाना, गुणा करना, भाग देना आदि।
Comparison	कोई मान बडा, छोटा, शून्य, Positive, Negative, बराबर है आदि।
Decision Masking	किसी Condition के आधार पर निर्णय लेना।
Logic	आवश्यक परिणाम को प्राप्त करने के लिए अपनाया जाने वाला Steps का क्रम।

केवल अंकों की गणना करना ही Processing नहीं कहलाता है। बल्कि किसी भी प्रकार के मान को जैसे कि किसी Document में से गलतियों को खोजने की प्रक्रिया या कुछ नामों के समूह को आरोही **(Ascending)** या अवरोही **(Descending)** क्रम में व्यवस्थित करने की प्रक्रिया को भी Processing ही कहते हैं।

Computer में Keyboard से जो भी Data Input किया जाता है, उस Data का तब तक कोई अर्थ नहीं होता है, जब तक कि Computer द्वारा उस Data पर किसी प्रकार की कोई Processing ना की जाए।

जैसे उदाहरण के लिए Computer में R, a, d, h, a ये पांच अक्षर अलग–अलग Input किए जाते हैं। इसलिए ये सभी अक्षर ***Row Data*** के समान हैं। Computer इन पांचों अक्षरों पर ***Processing*** करके इन्हें एक क्रम में व्यवस्थित कर देता है और हमें **"Radha"** नाम प्रदान करता है जो कि एक अर्थपूर्ण सूचना (***Information***) है।

Information – Processed Data

जिस Data पर Processing हो चुकी होती है, उसे **Processed Data** या **Information** कहते हैं। दूसरे शब्दों में कहें तो किसी Data पर Processing होने के बाद जो अर्थपूर्ण परिणाम (Result) प्राप्त होता है, उसे ही सूचना **(Information)** कहते हैं। एक Processing से Generate होने वाली किसी Information को हम किसी दूसरी Processing में फिर से Data के रूप में उपयोग में लेकर नई Information Generate कर सकते हैं और ये क्रम आगे भी जारी रखा जा सकता है।

उदाहरण के लिए R, a, m, K, i, l, l, e, d, R, a, v, a, n ये Characters हम अलग–अलग Input करते हैं। Computer पहले इन पर Processing करके Ram, Killed, व Ravan तीन शब्द बनाता है, जो कि हमारे लिए तीन अलग सूचनाओं को Represent करता है। क्योंकि **Ram. Ravan** व **Killed** तीनों ही शब्द अपने आप में परिपूर्ण हैं, इसलिए ये तीनों ही शब्द एक प्रकार की सूचना हैं जबकि यदि **"Ram Killed Ravan**" लिखा जाए तो इस वाक्य के लिए ये तीनों ही शब्द एक Data

के समान हैं, जो Processing के कारण आपस में एक व्यवस्थित क्रम में Arrange होकर एक सूचना प्रदान करते हैं।

सारांश में कहें तो Computer में हम सभी प्रकार की सूचनाओं को Data के आधार पर Store करते हैं। इन Data पर Processing करते हैं, जिससे सूचनाएं Generate होती हैं और इन सूचनाओं के आधार पर हम निर्णय लेते हैं। Data वास्तव में कोई अंक, अक्षर या चित्र हो सकता है। Computer में इन्हीं अंकों, अक्षरों या चित्रों को मानों के रूप में Manage किया जाता है। यानी Computer के सन्दर्भ में **Data** वास्तव में कोई मान या मानों का एक समूह होता है।

What is a Computer

Computer एक ऐसीElectronic Machine है, जो निर्देशों के समूह (जिसे **Program** कहते हैं) के नियंत्रण में Data या तथ्यों पर Processing करके **Information** Generate करता है।

Computer में Data को Accept करने और उस Data पर Required Processing करने के लिए किसी Program को Execute करने की क्षमता होती है। ये किसी Data पर Mathematical व Logical क्रियाएं करने में सक्षम होता है। Computer में Data को Accept करने के लिए Input Devices होती है, जबकि Processed Data यानी Information को प्रस्तुत करने के लिए Output Devices होती हैं। Data पर Processing का काम जिस Device द्वारा सम्पन्न होता है, उसे Central Processing Unit या CPU कहते हैं। ये एक Microprocessor होता है, जिसे Computer का दिमाग भी कहते हैं। किसी भी Computer निम्नलिखित क्षमताएं होती हैं:

1 User द्वारा Supplied Data को Accept कर सकता है।

2 Input किए गए Data को Computer की Memory में Store करके Required परिणाम प्राप्त करने के लिए किसी Instructions के समूह यानी किसी Program को Execute कर सकता है, जो कि उस Input किए गए Data पर Processing कर सकता है।

3 Data पर Mathematical व Logical क्रियाओं (Operations) को क्रियान्वित (Perform) कर सकता है।

4 User की आवश्यकतानुसार Output प्रदान कर सकता है।

Problem – Doing Something

Computer द्वारा हम किसी ना किसी प्रकार की समस्या का समाधान प्राप्त करने के लिए ही विभिन्न प्रकार के Programs लिखते हैं। इसलिए सबसे पहले हमें यही तय करना होगा कि आखिर हम Computer के संदर्भ में किस बात को एक समस्या के रूप में देख सकते हैं ?

यदि बिल्कुल ही सरल शब्दों में किसी समस्या को परिभाषित करें, तो Computer पर हम जिस किसी भी काम को Perform करके किसी प्रकार का कोई Result प्राप्त करना चाहते हैं, हम उस काम को समस्या के रूप में देख सकते हैं।

उदाहरण के लिए दो संख्याओं का योग **करना**, किसी परिणाम को Computer के Monitor पर Display **करना**, किसी भी प्रकार की कोई Calculation या Comparison **करना** आदि इन सभी कामों को हम समस्या के रूप में देख सकते हैं। यानी हम जो कुछ भी **करना** चाहते हैं, वह सबकुछ Computer के लिए एक समस्या ही है।

System – Interrelated Group of Units to Solve a Problem

Computer एक System होता है। जब किसी एक या एक से अधिक समस्याओं को सुलझाने या किसी लक्ष्य को प्राप्त करने के लिए कई स्वतंत्र इकाईयां (Individual Units) मिलकर काम कर रहे होते हैं, तो उन इकाईयों के समूह को **System** कहा जाता है।

जैसे कोई Hospital एक System होता है जिसे **Hospital System** कहा जाता है। Doctors, Nurses, चिकित्सा से सम्बंधित विभिन्न उपकरण, Operation Theater, Patient आदि किसी Hospital System की विभिन्न इकाईयां हैं। यदि इन में से किसी की भी कमी हो तो Hospital अधूरा होता है। इसी तरह से Computer भी एक System है, जिसके विभिन्न अवयव जैसे कि Monitor, Mouse, Keyboard, CPU व समस्या का समाधान प्राप्त करने से सम्बंधित Application Software आदि होते हैं और ये सभी आपस में मिलकर किसी समस्या का एक उचित समाधान प्रदान करते हैं।

DBMS

THE FUNDAMENTALS

DBMS – The Fundamentals

Computer में विभिन्न प्रकार के Data को Store व Manage करने के लिए कुछ Standard तरीकों को उपयोग में लाया जाता है, ताकि Computer द्वारा किसी भी समय Accurate व Up-To-Date Information को प्राप्त किया जा सके। जिन Standard तरीकों को उपयोग में लेकर किसी समस्या से सम्बंधित Data को Manage किया जाता है, उन तरीकों के समूह को ही **Database Management System** कहा जाता है।

किसी एक समूह से सम्बंधित सूचनाओं को कम से कम जगह में Store करने व Manipulate करने का सबसे अच्छा तरीका यही होता है कि उस "**Group Of Information**" को एक सारणी के रूप में Define किया जाए। एक सारणी किसी एक Group से सम्बंधित सूचनाओं को कम से कम जगह में व सबसे बेहतर तरीके Store करने का सबसे अच्छा तरीका होता है।

उदाहरण के लिए मानलो कि हमें किसी School के विभिन्न Students की Information को Computer पर Manage करना हो, तो हम एक सारणी बनाकर उसमें विभिन्न Students की जानकारियों को छोटे–छोटे टुकडों के रूप में निम्नानुसार Store कर सकते हैं:

```
//================================================================
```

Sr No	Name	Age	Sex	Class
123	Amit Sharma	15	Male	10
234	Rahul Varma	16	Male	10
121	Salini Bohra	15	Female	9
544	Silpa Roy	14	Female	8
534	Prince Mishra	13	Male	6
532	Devendra Bhati	14	Male	9

```
//================================================================
```

यदि हम इस सारणी में Represent किए गए सभी Data को एक साथ एक Group के रूप में देखें, तो हम कह सकते हैं कि जिस Student का Serial Number **123** है, उसका नाम Amit Sharma है और उसकी उम्र 15 साल है। साथ ही वह Class 10^{th} में पढता है।

इसी तरह से हम इस सारणी में Represent किए गए अन्य Students की भी विभिन्न प्रकार की जानकारियों को प्राप्त कर सकते हैं। Database Management System में इसी तरीके को उपयोग में लिया जाता है व समस्या से सम्बंधित इकाई की जिन जानकारियों को Computer द्वारा Manage करना होता है, उन जानकारियों को छोटे–छोटे टुकडों में Divide करके, उन्हें Logical Tables में Data के रूप में Store कर लिया जाता है।

जब हम Computer द्वारा किसी समस्या को Solve करना चाहते हैं, तब सबसे पहले हमें उस समस्या से सम्बंधित उन मुख्य Entities को Identify करना होता है, जिन्हें हम Computer पर Mange करना चाहते हैं। उदाहरण के लिए यदि हम किसी Student से सम्बंधित विभिन्न प्रकार की

Information को Computer द्वारा Manage करना चाहते हैं, तो इस समस्या के लिए Student वह Entity होता है, जो कि हमारी समस्या से सम्बंधित होता है।

किसी समस्या में हमेंशा केवल एक ही Entity हो, ऐसा कभी भी जरूरी नहीं होता है। विभिन्न प्रकार की परिस्थितियों में किसी समस्या से सम्बंधित एक से ज्यादा प्रकार के Entities हो सकते हैं। दुनियां का कोई भी व्यक्ति, वस्तु या घटना किसी विशेष परिस्थिति में एक Entity के रूप में Identify हो सकता है।

जब एक Programmer किसी Business से सम्बंधित Data को Computer पर Manage करना चाहता है, तब वह जिस Organization के लिए Application Develop कर रहा होता है, उस Organization के आधार पर ये तय होता है कि उस Application से सम्बंधित मुख्य Entities कौन–कौन से हैं।

चूंकि विभिन्न प्रकार के Organizations विभिन्न प्रकार के काम करते हैं, इसलिए किसी एक Organization के लिए जो व्यक्ति, वस्तु या घटना एक Entity के रूप में Represent होता है, वही व्यक्ति, वस्तु या घटना किसी दूसरे Organization के लिए भी एक मुख्य Entity हो, ऐसा जरूरी नहीं होता है।

उदाहरण के लिए किसी School में Teaching करवाने वाला Teacher उस School के Database Application के लिए एक **Employee Entity** होता है, लेकिन जब वही Teacher किसी Bank में अपना Account Open करवाता है, तब उस Bank के लिए वही Teacher एक **Customer Entity** हो जाता है।

जब हम Computer में किसी Entity को Represent करना चाहते हैं, तब हमें उस Entity की उन Characteristics को Identify करना होता है, जिन्हें हम Computer पर Manage करना चाहते हैं। ये Characteristics ही उस Entity को Computer में Represent करने के माध्यम होते हैं।

दुनियां के हर Object की अपनी कुछ विशिष्टताएं या लाक्षणिकताएं (Characteristics) होती हैं, जिनके कारण हम उस Object को किसी दूसरे Object से अलग पहचान पाते हैं। चूंकि किसी समस्या से सम्बंधित Entity भी इसी Real World का कोई ना कोई Object होता है, इसलिए उस Entity की भी अपनी कुछ Special Characteristics होती हैं, जिनसे उस Entity को Identify किया जा सकता है। Entity की इन Characteristics को सामान्यतया **Attributes** कहा जाता है और ये Attributes ही वे माध्यम होते हैं, जिनके द्वारा हम समस्या से सम्बंधित Entity को Computer में Represent करते हैं।

किसी भी Entity का **Attribute.** Information का वह सबसे छोटा हिस्सा होता है, जिसे Computer पर Store व Manage किया जाना होता है। इस Attribute को सामान्यतया **Field** कहा जाता है। इन Fields में हमेंशा किसी ना किसी प्रकार का मान यानी Data Store किया जाता है।

किसी समस्या से सम्बंधित किसी एक Entity के जिन Attributes को Computer पर Mange करना होता है, उसी समस्या में उसी प्रकार के बहुत सारे Entities के लिए भी उन्हीं Attributes को Computer पर Manage करना होता है। इस स्थिति में समान Group के ढेर सारे Entities समान Attributes को Share करते हैं, जिन्हें Computer में Field द्वारा Represent किया जाता है। किसी समान Field को Share करने वाले सभी Entities के Group को **Entity Set** कहा जाता है।

अब हम एक उदाहरण द्वारा इस पूरी प्रक्रिया को समझने की कोशिश करते हैं। मानलो कि किसी School का Principal उसके School में पढने वाले सभी Students की जानकारियों को Computer पर Maintain करना चाहता है, ताकि जब भी उसे किसी Particular Student से सम्बंधित जानकारियों की जरूरत हो, वह उस Student का Serial Number उस **Student Database Application Software** Input करे और Computer उस Student से सम्बंधित सभी Information को Screen पर Display कर दे।

किसी भी Database System Application को Develop करने से पहले हमें सबसे पहले समस्या को अच्छी तरह से Analyze करके ये पता लगाना होता है कि आखिर उस System की मुख्य आवश्यकता क्या है और उस आवश्यकता को पूरा करने से सम्बंधित कुल कितने Entities हैं व वे Entities कौन–कौन से हैं? चूंकि हमारी इस समस्या को यदि हम ध्यान से देखें तो इस समस्या की मुख्य Requirement School के Students की **Information** ही है और समस्या से सम्बंधित मुख्य Entity भी **Student** ही है।

समस्या से सम्बंधित Entity का पता चल जाने के बाद हमें ये पता लगाना होता है, कि उस Entity से सम्बंधित किन बातों को Computer पर Maintain करना है। ये बातें ही उस Entity का Attributes होते हैं, जिन्हें Fields के रूप में Define किया जाता है।

चूंकि हमारी इस समस्या में मुख्य Entity Student है और एक Student से सम्बंधित वे जानकारियां जिनका उपयोग एक School में किया जाता है, समस्या से सम्बंधित जानकारियां हैं। किसी Student की मुख्यतः निम्न जानकारियां हो सकती हैं, जिन्हें एक Computer पर Maintain करना School के Principal के लिए उपयोगी हो सकता हैः

//==========

1. Student का नाम
2. Student के पिता का नाम
3. Student का Address
4. Student की City
5. Student का जिला
6. Student की Class
7. Student की Date of Birth
8. Student की School में Join करने की Date of Admission
9. Student की Age

```
   10 Student का Serial Number
//=================================================================
```

ये Description किसी भी Student की Information के उन छोटे–छोटे टुकडों (**Attributes**) को Represent करते हैं, जिनकी School के Principal को जरूरत हो सकती है। इन Descriptions के आधार पर हमें निम्नानुसार विभिन्न Fields प्राप्त हो सकते हैं:

```
//=================================================================
```

1 SerialNumber
2 Name
3 FName
4 Address
5 City
6 District
7 Class
8 DateOfBirth
9 DateOfAdmissiona
10 Age

```
//=================================================================
```

यदि हम इन जानकारियों के टुकडों को Combined रूप में देखें, तो ये सभी Files आपस में मिलकर किसी एक Student से सम्बंधित उन जानकारियों को Represent करते हैं, जिनकी एक School के Principal को जरूरत हो सकती है। ये सभी Fields हमारी समस्या से सम्बंधित Entity के उन Attributes को Represent करते हैं, जिन्हें Computer पर Store व Mange किया जाना है। यदि हम इन Fields को Title के रूप में Specify करें व इनके नीचे इनमें Store किए जाने वाले मानों (Data) को Specify करें, तो हमें निम्नानुसार Format प्राप्त हो सकता है, जो कि एक प्रकार की सारणी है:

SrNo	Name	FName	City	Dist.	Class	DOB	DOA
001	Rahul	Mohan Lal	Falna	Pali	10	10-02-1982	15-7-1987
002	Rohit	Sohan Lal	Bali	Pali	09	11-12-1983	05-7-1987
003	Krishna	Gopal	Desuri	Pali	08	20-03-1981	10-7-1987
004	Madhav	Ram Lal	Falna	Pali	10	30-2-1982	01-7-1987
005	Achyut	Nand Lal	Desuri	Pali	07	12-12-1986	13-7-1987
006	Manohar	Rohan Lal	Bali	Pali	10	10-11-1982	15-7-1987

इन जानकारियों के अलावा भी Student की विभिन्न प्रकार की अन्य जानकारियों को भी Store करके Manage किया जा सकता है। हम देख सकते हैं कि विभिन्न Students समान Attributes को Share कर रहे हैं, इसलिए Students के इस समूह को Entity Set कहते हैं। हमारे Entity Set में कुल 6 Students हैं। इस सारणी का हर Column किसी Student के किसी एक Attribute के मान को Represent कर रहा है।

उदाहरण के लिए Name Column हर Student का केवल नाम Specify करता है, इसी तरह से DOB Column हर Student का **Date Of Birth** Specify कर रहा है। ये विभिन्न Columns किसी Student Entity के विभिन्न **Attributes** या **Fields** को Specify कर रहे हैं।

इस सारणी के आधार पर यदि हम SrNo 001 वाले Student की जानकारी प्राप्त करना चाहें, तो हमें Left To Right चलते हुए इस Serial Number वाले Student का नाम Rahul प्राप्त होता है, जिसके पिता का नाम Mohan Lal है और वह Falna नाम की City में रहता है। इस City का District Pali है और वह Class 10^{th} में पढता है। Rahul की Date Of Birth **10 Feb 1982** है और उसने इस School में **15 July 1987** को Admission लिया है। यानी हम इस सारणी के आधार पर कह सकते हैं कि इस सारणी के सभी Fields आपस में Logically Related हैं, इसी कारण से सभी Fields आपस में मिलकर किसी एक Student से सम्बंधित सभी जानकारियां प्रदान कर रहे हैं।

जब बहुत सारे Fields जो कि आपस में Logically Related हों, मिलकर किसी एक Entity से Related विभिन्न प्रकार की जानकारियां Provide करते हैं, तो Logically Related Fields के इस Group को एक **Record** कहा जाता है। यानी यदि हम पिछली सारणी के आधार पर कहें तो इस सारणी का हर Row एक Unique Student के Record को Specify कर रहा है। किसी Record को Database Management System की भाषा में **Tuple** कहा जाता है।

जब किसी समस्या में एक से अधिक Entities Involved होते हैं, तब उन सभी Entities के Attributes को इसी प्रकार से प्राप्त किया जाता है और इसी प्रकार से एक Table द्वारा हर Entity को Represent किया जाता है।

सारांश में कहें तो हम कह सकते हैं कि किसी समस्या से सम्बंधित विभिन्न Entities की जिन विशेषताओं को Computer में Store करना होता है, उन विशेषताओं को Entity का Attribute कहा जाता है, जिसे Database Management System की भाषा में **Field** कहा जाता है। किसी Entity के Attributes (**Data Fields**) का वह समूह जो कि आपस में Logically Related होते हैं, किसी एक Entity से सम्बंधित विभिन्न सूचनाओं को Specify करते हैं। इन Logically Related Fields के समूह को **Record** कहा जाता है, जो किसी Entity के उन Data को Specify करता है, जिन्हें Computer पर Store व Manage किया गया होता है।

जब एक ही प्रकार के बहुत सारे Entities यानी Entity Set के Data को Computer पर Store व Manage किया जाता है, तब इस Entity Set के समूह को **Table** या **Entity** या Database Mange System की भाषा में **Relation** हो जाता है। यानी सरल शब्दों में कहें तो Fields के समूह को Record कहते हैं। Records के समूह को **Table** कहते हैं और Tables के समूह को Database कहते हैं।

Requirement of Good Database Design

आज जितने भी Businesses Database System पर निर्भर हैं, यानी अपने Business से सम्बंधित जानकारियों को Computer द्वारा Manage करते हैं, वे सभी Accurate व Up-To-Date Information प्राप्त करने के लिए ही Computer का उपयोग करते हैं। जितने भी Business Corporations Computer पर अपने Business से सम्बंधित Data को Maintain करते हैं, उन सभी को कभी ना कभी किसी ना किसी रूप में अपने Data के Report की जरूरत होती है, जिसके आधार पर उस Business को Operate करने वाला Authorizer अपने व्यवसाय से सम्बंधित जरूरी निर्णय लेता है।

इसलिए ये जरूरी हो जाता है कि किसी भी Database में Store किए जाने वाले Data **Accurate, Complete** व इस तरह से **Well Organized** होने चाहिए, ताकि जब भी किसी प्रकार के Information की जरूरत हो और जिस Format में Information की जरूरत हो, उस Information से सम्बंधित Data को उसी Format में **Fastly** व **Accurately** प्राप्त किया जा सके।

किसी भी Database System को Develop करते समय सबसे आधारभूत तथ्य के रूप में इसी बात का ध्यान रखा जाना होता है, कि Develop किया जाने वाला Application चाहे **Local Area Network** पर Use किया जाना हो या किसी Web Site से Data को Access किया जाना हो, दोनों ही स्थितियों में Database से प्राप्त होने वाला Data Accurate व Fast होना चाहिए। यानी Database चाहे छोटा हो या बडा, यदि हम एक Database System को बिना किसी परेशानी के लम्बे समय तक के लिए उपयोगी बनाना चाहते हैं, तो हमें Database को बहुत ही सावधानीपूर्वक अच्छे तरीके से Design करना जरूरी होता है।

यदि Database का Design कमजोर हो, तो चाहे जितना भी अच्छा Program Develop कर लिया जाए, उस Database System से पैदा होने वाली परेशानियों से बचा नहीं जा सकता है। किसी Database Management System से सम्बंधित Application में जितनी भी परेशानियां पैदा होती हैं, उनमें से ज्यादातर परेशानियों का कारण Database का खराब Design ही होता है।

जब किसी Database System Application को Develop करते समय Database के Design पर सावधानीपूर्वक ज्यादा ध्यान नहीं दिया जाता है, तब भविष्य में उस Application से सम्बंधित विभिन्न प्रकार की परेशानियों का सामना करना पडता है।

अच्छे Database Design का मतलब ये है कि हम Database System को Develop करते समय पर्याप्त समय लें व सावधानीपूर्वक Database को इस तरह से Design करें, जो कि भविष्य में कम से कम परेशानी पैदा कर सके।

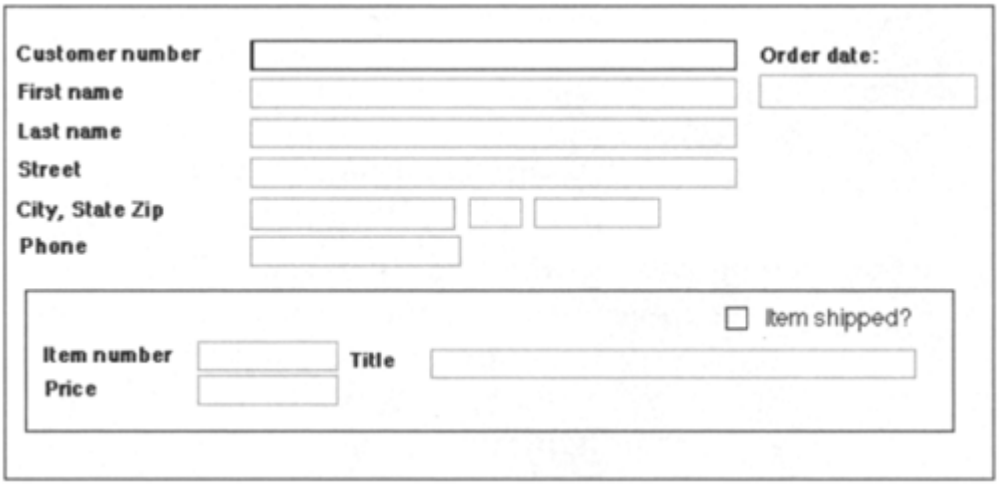
इस प्रकार का Database Design करते समय हमें इस बात पर Focus रखना होता है, कि हम जिस Organization से सम्बंधित Database System Develop कर रहे हैं, उस Organization में विभिन्न प्रकार के कामों को किस प्रकार से पूरा किया जाता है।

यदि Organization जिस तरीके से काम करता है, उस तरीके से Organization को भविष्य में किसी तरह की परेशानी का सामना नहीं करना पडता है, तो निश्चित रूप से उस Organization के काम करने के तरीके के आधार पर Develop किया गया Database System भी भविष्य में किसी प्रकार की कोई परेशानी पैदा नहीं करेगा।

Effects of Poor Database Design

एक Database का Design किस प्रकार से किसी Database System Application में एक बहुत ही महत्वपूर्ण Roll अदां करता है, इस बात को हम एक सामान्य से Business Example द्वारा ही समझ सकते हैं। हम जो Business Example ले रहे हैं, उसमें एक बहुत ही Poor Design को Use किया गया है और इसी Poor Design के कारण विभिन्न प्रकार की परेशानियां Generate होती हैं, जिन्हें एक–एक करके समझाया गया है।

इस Business Example को हमने "**Music Store**" नाम दिया है। इस Business में एक Music Store विभिन्न Titles के CDs व DVDs की Selling का काम करता है। ये Music Store **Mail** से आने वाले Orders के आधार पर Titles Selling का काम करता है। इस Business Example में जब भी कोई Customer किसी Single Item को Purchase करने के लिए Order देता है, Music Store का एक Employee निम्न Form को Fill करके Customer के Order को Computer पर Data के रूप में Store या Record कर लेता है।



चूंकि हम ये मान रहे हैं कि Music Store पर आने वाले Orders Mail द्वारा आते हैं, इसलिए एक ही City से कई Customers Orders आ सकते हैं। इस **Music Store** Software में हर Customer को Uniquely Identify करने के लिए एक Unique Number Assign किया गया है। हर Customer का Unique Customer Number Create करने के लिए हर Customer के Pincode

Number के साथ उसके नाम के पहले तीन Character को Use किया जाता है और नाम के बाद तीन Digit का एक Sequence Number Specify किया जाता है।

यानी यदि Krishna नाम का कोई Customer किसी Item के लिए इस Music Store पर कोई Order देता है और वह Customer **123456** Pincode Number वाले शहर में रहता है, तो उस Customer को Identify करने के लिए बनने वाला Customer Code **123456KRI001** होगा। Sequence Number इस बात की पुष्टि करता है कि एक ही शहर में रहने वाले एक ही नाम के दो Customer को भी Music Store द्वारा Uniquely Identify किया जा सकता है।

जब Music Store में Titles Distributor से कोई नया Title आता है, तो Music Store का एक Employee उन सभी Customers को Search करता है, जिन्होंने उस Title के लिए पहले से ही Order दे रखा था। फिर वह Employee उन Customers की Computer में Stored **Order Data** के आधार एक List Create करता है और Form पर स्थित "**Item shipped?**" Check Box में एक '**X'** Place कर देता है, जो इस बात का Signal होता है कि पहले से आए हुए Order को पूरा कर दिया गया है।

पहली नजर में देखने पर Music Store का ये Management काफी साफ सुथरा व सरल लगता है। थोडे समय तक ये Software काम भी ठीक तरह से करता है। लेकिन एक–दो साल बाद इस Software से Serious Problems पैदा होने लगती हैं।

Unnecessary Duplicated Data and Data Consistency

Music Store Database में बहुत सारा Data बार–बार अनावश्यक रूप से Duplicated Form में Input करना पडता है और एक ही प्रकार के Data को बार–बार किसी Database में Store करने से विभिन्न प्रकार की समस्याएं पैदा होती हैं, जिससे Database को Manage करना कठिन हो जाता है। किसी Database में एक ही प्रकार के Data को बार–बार Store करने की प्रक्रिया को **Data Redundancy** कहा जाता है।

जब भी कोई Customer Music Store पर कोई Title Order करता है, उपरोक्त Form में उस Customer के Order को Record कर लिया जाता है। किस Customer ने कौनसा Order Place किया है, इस बात की जानकारी रखने के लिए Order देने वाले Customer का नाम, Address व Phone Number भी Order की Information के साथ ही Database में Store कर लिया जाता है।

अब चूंकि एक ही Customer एक से ज्यादा बार Order कर सकता है, इसलिए कोई एक ही Customer जितनी बार भी किसी Title के लिए Music Store पर Order देता है, हर बार उस Customer के नाम, Address व Phone Number को Form पर Fill करके Order की Information के साथ Database में Store कर लिया जाता है। जिससे उस Customer की Information का बार–बार Duplication होता है।

जब हमारे Database में इस तरह से Duplicated Form में Data Store हो रहे होते हैं, तब हमें इस बात का ध्यान रखना जरूरी हो जाता है कि Duplication Form में Store होने वाले सभी Data हर बार समान रूप में ही Database में Store हों।

दूसरे शब्दों में कहें, तो किसी Customer द्वारा दिए जाने वाले हर Order को Music Store Application में Store करते समय हमें इस बात का ध्यान रखना जरूरी होता है कि उस Customer के हर Order में उसका नाम, Address व Phone Number एक जैसे ही Store किए जाए।

इसी तरह से एक Single Title के लिए जितने भी Order Place किए जाते हैं, उन सभी Orders में Title को एक भी Character के हेर–फेर के बिना एक जैसा Type करना जरूरी होता है। इसी पूरी प्रक्रिया के कारण Input किया जाने वाला Duplicated Data Consistent (विश्वसनीय) हो जाता है।

जैसे–जैसे Database का Data बढता जाता है, इस प्रकार की Constancy को Maintain करना काफी मुश्किल हो जाता है। ज्यादातर Business Oriented Database Software Case Sensitive होते हैं, जिनमें Uppercase Letters व Lowercase Letters अलग–अलग Behave करते हैं।

हम ये मान सकते हैं कि Music Store Form में Order की Entry करने वाला कोई भी Operator इतना Perfect Typist नहीं हो सकता, जो हमेंशा इस बात को ध्यान रख सके कि उसने किस Customer के नाम व पते में कौनसा Character Capital Letter में लिखा था और कौनसा Character Small Letters में।

इस स्थिति में किसी एक ही Customer के Orders की Entry करते समय यदि एक भी Character के Typing का Difference हो गया, तो Database Software एक ही Customer के दो अलग Unique Record Create कर सकता है।

उदाहरण के लिए मानलो कि “Rahul” व “Rohit” नाम के दो Customers “UMI 10” नाम के Title की DVD का Order Music Store पर Place करते हैं। Typist जब इन दोनों Orders को Music Store Application के Form द्वारा Database में Store करता है, तब वह “Rahul” का Order Specify करते समय Title के स्थान पर “UMI 10” Character Combination का प्रयोग करता है, जबकि “Rohit” का Order Specify करते समय Title के स्थान पर “UMI10” Character Combination का प्रयोग करता है।

अब मानलो कि Music Store को उन Customers की जानकारी प्राप्त करने की जरूरत पडती है, जिन्होंने “UMI 10” नाम के Album का Order दिया है। Music Store Software से इस बात की जानकारी प्राप्त करने के लिए यदि Typist “UMI10” Character Combination का प्रयोग करता

है, तो Application द्वारा Generate होने वाली Pending Orders की List में "Rahul" का Order Display नहीं होगा और यदि Typist "UMI 10" Character Combination का प्रयोग करता है, तो "Rohit" का Order Display नहीं होगा।

इस स्थिति में हम समझ सकते हैं कि "Rahul" व "Rohit" दोनों में से किसी एक के Pending Order की ही जानकारी ये Music Store Application दे सकता है। जिससे किसी ना किसी Customer के Order की Request तो अधूरी ही रहेगी।

हमारे Current "Music Store" Application में इस बात को Ensure करने की कोई व्यवस्था नहीं है कि Database में Data भले ही Duplicated Form में Store हों, लेकिन Duplicated Data भी Consisted Form यानी विश्वसनीय रूप से Database में Enter होंगे और उपरोक्त प्रकार की समस्याएं Generate नहीं होंगी। इस प्रकार की समस्याओं को Solve करने के लिए दो तरीके उपयोग में लिए जा सकते हैं:

पहला तरीका ये है कि Data Duplication को जितना हो सके उतना रोका जाए। इस Solution के बारे में हम आगे और अच्छी तरह से समझेंगे। लेकिन यहां ये जान लेना जरूरी होगा कि किसी भी Database Application में Data की Redundancy यानी Duplication को पूरी तरह से Eliminate करना ना तो सम्भव है और ना ही इसकी जरूरत होती है। यानी हर Database में थोडा बहुत Data Duplication तो होता ही है, जो कि किसी भी Database को ठीक से Manage करने के लिए जरूरी भी होता है।

दूसरा तरीका ये है कि जब भी किसी Order की Entry Database में हो, तो कोई ऐसा तरीका होना चाहिए, जो ये Verify कर सके कि जब Data Duplicate Form में Enter हो, तब Data हमेंशा एक ही प्रकार से Database में Store हो।

एक Well Design Database में इन दोनों Solutions को Use किया जाता है। Duplication से पैदा होने वाली दूसरी समस्या ये है कि एक ही Information को बार–बार Store करने से Database की Size बढ जाती है, क्योंकि एक ही प्रकार के Data, Store होने के लिए Memory में बार–बार Storage Space लेते हैं।

लेकिन चूंकि आज Disk Space उतनी महंगी नहीं है, जितनी पहले हुआ करती थी, इसलिए आज Storage Space की बचत के लिए Redundant Data को Eliminate करना उतना बडा कारण नहीं है, जितना Database को सरलता से Maintain करना।

Data Insertion Problems

Music Store जैसे जितने भी Database होते हैं, जैसे कि Publication आदि, इनमें जितने भी Titles होते हैं, उनकी एक List बनती है, जिसके द्वारा ये पता चलता है कि उस Music Store या Publication पर किन–किन Titles के Items उपलब्ध हैं। उदाहरण के लिए जब भी किसी नए

Title की CD/DVD Market में आने वाली होती है, इन Music Store पर उस नए Title को अपने Catalog में Add करना होता है।

ठीक इसी तरह से किसी Publication में जब भी कोई नई Book Publish होती है, उस Publication को भी अपने Catalog को Update करना पडता है। ऐसा इसलिए किया जाता है ताकि Music Store या Book Store के Customers इस नए Title को Advance में Order कर सकें।

Catalog एक ऐसी List होती है, जिसमें कोई Music Store या Publication अपने Items की List को Store करता है, ताकि वह अपने Customers को इस बात की जानकारी दे सके कि उसके पास कौन–कौन से Title के Item उपलब्ध हैं।

चूंकि हम जिस Music Store Application को उदाहरण के रूप में उपयोग में ले रहे हैं, उसमें विभिन्न Titles के Catalog को Maintain करने की कोई व्यवस्था नहीं है, इसलिए जब भी Market में किसी नए Title के आने की सूचना मिलती है, इस Music Store में एक Employee स्वयं अपने Catalog को Update करता है और अपने Titles के Updated Booklet को अपने हर Customer को भेज देता है, ताकि उनका Customer ये तय कर सके कि उसे कौन–कौन से Title Order करने हैं। इस Catalog Booklet में बहुत सारे Pages हो सकते हैं और हर Page को Music Store का कोई Employee स्वयं “Copy Paste” की प्रक्रिया द्वारा तैयार करता है।

अब मानलो कि हम ये चाहते हैं कि ये Catalog Booklet Database के आधार पर स्वयं ही तैयार हो जाए। चूंकि विभिन्न प्रकार के Titles Database में Stored होते हैं, इसलिए हम Catalog Booklet को Database के आधार पर तैयार कर सकते हैं। लेकिन फिर भी हम Current Database के आधार पर ये काम नहीं कर सकते हैं। इसके दो कारण हैं:

किसी भी Catalog में किसी भी Title से सम्बंधित विभिन्न प्रकार की Additional जानकारियां होती हैं। उदाहरण के लिए किसी Title के Singer, Music Director, Financer आदि की जानकारियां हो सकती हैं और Title से सम्बंधित कुछ Extra Description हो सकती हैं।

चूंकि हमारे Music Store Database में इन जानकारियों को Store करने की व्यवस्था नहीं है, इसलिए इस समस्या के समाधान के रूप में हम हमारे Music Store के Database को Modify करके उसमें नए Fields Create कर सकते हैं। लेकिन ये तरीका भी पूरी तरह से Catalog Create करने में सक्षम नहीं हो सकता। क्योंकि एक ही Title को बहुत सारे Customers Order कर सकते हैं।

इस स्थिति में हर Order के साथ Title की विभिन्न Descriptions को Computer में Store करने से Data की Redundancy बढ जाएगी और यदि केवल एक ही Customer के Order में किसी Title की Information को Store किया जाए तो हमेंशा इस बात को ध्यान रखना होगा कि किस Title की Extra जानकारियों को किस Customer के Order में Specify किया गया है, जो कि

एक नामुमकिन काम है।

दूसरी समस्या ये है कि हमारे Music Store Application के Database Management System में ऐसी कोई व्यवस्था नहीं है, जिससे किसी Title को Advance में ही Database में Enter किया जा सके। जब तक कोई Customer उस नए Title का Order Place नहीं करता है, तब तक उस Title को Database में Store नहीं किया जा सकता है और जब तक Title Database में Store नहीं होगा, तब तक Updated Catalog Booklet Create नहीं किया जा सकता।

साथ ही Customer तब तक उस Title का Order Place नहीं कर सकता जब तक कि उसे Updated Catalog Booklet प्राप्त ना हो, क्योंकि उसे नए Title की जानकारी Updated Catalog Booklet से ही प्राप्त होती है। इस समस्या को Database Management की भाषा में "**Insertion Anomaly**" कहा जाता है।

Data Deletion Problems

हमारे इस Music Store Database Program से जब किसी Data को Delete किया जाता है, तब भी कुछ समस्याएं सामने आती हैं। मानलो कि एक Customer केवल एक Item का Order देता है। Order को Process करने के बाद यानी Order की Entry Music Store के Form द्वारा Database में कर देने के बाद पता चलता है कि उस Item को Manufacture करने वाले Manufacturer ने उस Item को Create करना बन्द कर दिया है। इस स्थिति में Music Store अपने किसी भी Customer को वह Item नहीं भेज सकता, जिसने इस Out Of Stock Item के लिए Order किया है।

अब चूंकि Orders की Entry Database में पहले होती है, इसलिए उन सभी Customers के Order की List में से इस Item का Reference Database से Delete करना होगा, जिन्होंने उस Particular Item के लिए Order किया है, जो कि अब Available नहीं है।

चूंकि जिन लोगों ने बहुत सारे Items Order किए हैं, उनके Order की List में से केवल इस Unavailable Item के Reference को Delete करना होगा, जबकि जिन लोगों ने केवल इसी Item का Order किया था, Database से उनके Order की List में से इस Item का Reference Delete करने पर उनके Order में कोई Item नहीं बचेगा, जिसे उस Customer को भेजा जा सके। इस स्थिति में ऐसे Customers का Order भी Delete कर दिया जाएगा।

अब चूंकि विभिन्न Customers की Information भी उनके Order के साथ ही Database में Store होती है, इसलिए यदि किसी Customer ने उस Unavailable Item के लिए Music Store को First Time Order दिया हो, तो जब उस Customer का Order Delete होगा, उस Order के साथ ही उस Customer की Information भी Delete हो जाएगी।

इस स्थिति में Music Store उस Customer को भविष्य में कोई Catalog Booklet नहीं भेज

सकेगा और Music Store का उस Customer से Link ही टूट जाएगा, क्योंकि उस Customer का Address उसके Un-Fulfilled Order के साथ ही Delete हो चुका है। Database की इस Problem को Database की भाषा में "**Deletion Anomaly**" कहते हैं।

Meaningful Identifiers

हमारे Database में एक और बडी समस्या है जो कि Customer को Uniquely Identify करने के लिए Create किए जाने वाले Customer Number की है। इस Database में किसी भी Customer को Uniquely Identify करने के लिए एक विशेष तरीके को Use किया गया है, जिसमें उस Customer के नाम व City के Pincode Number को Use किया जाता है।

अब मानलो कि एक Customer जिस City में रहता है, उस City को छोडकर किसी दूसरे शहर में चला जाता है। इस स्थिति में उसी Customer को Identify करने के लिए फिर से एक नया Customer Number Create किया जाएगा, जो कि उस दूसरी City से सम्बंधित होगा। जिससे हमारे इस Database में एक ही Customer के दो ID हो जाएंगे जो एक ही Customer को Refer करेंगे।

मानलो कि एक Customer Music Store पर एक Order Place करता है और उसके बाद वह अपनी City Change कर लेता है। अब यदि वह Customer अपनी City Change कर लेने के बाद ये जानना चाहता है कि उसके कितने Order Music Store पर Pending हैं, जिसके Items को Music Store ने उस Customer को Serve नहीं किया है। उस Customer के Pending Orders की जानकारी प्राप्त करने के लिए Music Store का वह Operator जो कि Music Store Software को Operate करता है, उस Customer से उसका Customer Number पूछेगा, ताकि वह उस Customer Number वाले Customer की कुल Transactions की List प्राप्त कर सके।

चूंकि Customer अब दूसरे शहर में रहता है, इसलिए उसका Customer Number Change हो गया है। इस स्थिति में वह अपने Current Customer Number की जानकारी उस Operator को देगा। जिसका मतलब ये है कि इस शहर में आने से पहले उसने Music Store पर जितने भी Order Place किए हैं, उनकी जानकारी उसे उसके Current Customer Number द्वारा प्राप्त नहीं होगी, क्योंकि वे Orders उसने अपने पुराने शहर से दिए थे और उस शहर में रहने के कारण उसका Customer Number दूसरा था। इस स्थिति में वह Customer ये मान सकता है कि उसका Order Music Store को प्राप्त नहीं हुआ। इसलिए वह Customer उसी Order को फिर से Music Store पर Place कर देगा।

इस स्थिति में एक ही Customer के एक ही Order की दो Entry Music Store Database में हो जाएगी और जब उन दोनों Orders को Music Store द्वारा पूरा किया जाएगा, तब एक ही Customer को समान Items की दो Copies प्राप्त हो जाएंगी, जिसकी दूसरी Copy को सम्भवतया

वह Customer फिर से Music Store को Return कर देगा और Music Store को Transportation Charges स्वयं वहनं करने होंगे।

Entities and Data Relationships

Database वह स्थान होता है, जिसमें Data को ना केवल Store किया जाता है, बल्कि उन Store होने वाले Data के बीच की आपसी Relationship की Information को भी Store किया जाता है। Database के Concept का मुख्य आधार ये है कि किसी समस्या से सम्बंधित जानकारियों को चाहे बहुत सारे User Access कर रहे हों या फिर चाहे एक User Access कर रहा हो, User को इस बात की चिन्ता करने की जरूरत नहीं होती है कि समस्या से सम्बंधित विभिन्न प्रकार के Data Computer में किस प्रकार से Store हो रहे हैं। User अपने Database से विभिन्न प्रकार के Data को केवल उनकी Relationship के आधार पर Access करके विभिन्न प्रकार की Database Related Information Generate करता है।

हालांकि User अपने Data को Logically Access करता है जबकि Data वास्तव में Physically Store होते हैं। इसलिए User व Database के बीच के आपसी Conversation को एक दूसरे Form में Translate करके एक दूसरे को Available करवाने का काम एक Software करता है, जिसे *Database Management System Software* (**DBMS**) कहा जाता है।

हम जिस Formal Way को Use करके विभिन्न प्रकार के Data Relationship किसी DBMS Software को Express करते हैं, उस Formal तरीके को Data Model कहा जाता है। हम जिस Relational Data Model को इस पुस्तक में पढेंगे, वह केवल एक Formal Structure ही होता है।

इससे पहले कि हम किसी Database को Design करें, हमें Store किए जाने वाले विभिन्न प्रकार के Data के बीच की आपसी Relationships को Identify करना होता है। सामान्यतया विभिन्न प्रकार के DBMS Softwares केवल एक ही Data Model को Support करते हैं। इसलिए जब हम किसी DBMS Software को Choose कर रहे होते हैं, तब हम वास्तव में अपने Data Model को भी Choose कर रहे होते हैं।

Entities and Their Attributes

वह चीज जिसकी जानकारियों को हम Data के रूप में किसी Database में Store करते हैं, Entity कहलाता है। हमारे Music Store Application के सम्बंध में Customer एक प्रकार का Entity है क्योंकि हम Customer से Related Data को Database में Store करते हैं। Entity हमेंशा कोई Physical वस्तु ही हो, ऐसा जरूरी नहीं होता है।

दुनियां की किसी भी उस वस्तु को भी हम Entity मान सकते हैं, जो कि किसी Physical Entity से

Related हो। उदाहरण के लिए किसी Bank का Account भी एक Entity हो सकता है, क्योंकि वह एक Physical Person से ही सम्बंधित होता है।

हर Entity के कुछ Data होते हैं, जो उस Entity को Describe करते हैं। Entity के इन Data को Entity का Attribute कहा जाता है। उदाहरण के लिए हमारे Music Store Application के सम्बंध में एक Customer को उसके Customer Number, First Name, Last Name, Street, City, State, Pincode व Phone Number द्वारा Describe किया जाता है। इसलिए ये सभी Data Customer Entity के Attributes हैं।

जब हम किसी Database में किसी Entity को Represent कर रहे होते हैं, तब वास्तव में हम किसी Entity के केवल Attributes को ही Computer में Store कर रहे होते हैं। Attributes का हरेक Group जो कि किसी एक Single Real World Entity के एक उदाहरण को Describe करता है, वही Attributes का Group उस Entity के दुनियां कि किसी भी अन्य Instance को Represent कर सकता है।

यानी जो Attributes किसी Rahul नाम के Customer को Represent कर सकते हैं, वे ही Attributes Rohit नाम के किसी दूसरे Customer को भी Represent कर सकते हैं। उदाहरण के लिए Students Entity की निम्न सारणी में चार Student Instance हैं और सभी Instance समान Attributes को Share कर रहे हैं।

```
//=================================================================
SrNo  Name     FName          Add     Dist.  Class  DOB         DOA
//=================================================================
001   Rahul    Mohan Lal      Falna   Pali   10     10-02-1982  15-7-1987
002   Rohit    Sohan Lal      Bali    Pali   09     11-12-1983  05-7-1987
003   Krishna  Gopal          Desuri  Pali   08     20-03-1981  10-7-1987
004   Madhav   Ram Lal        Falna   Pali   10     30-2-1982   01-7-1987
//=================================================================
```

यदि हमारे इस Database में 2000 Students होते, तो इस सारणी द्वारा 2000 Students Attributes के Collections होते।

Entity Identifiers

किसी Entity को Describe करने वाले विभिन्न Data को Database में Store करने का मुख्य Purpose यही है कि इन्हें बाद में किसी Information को Retrieve करने के लिए Use किया जाएगा।

इसका मतलब ये हुआ कि हमें किसी ना किसी तरीके से किसी एक Entity को किसी दूसरे Entity से अलग Represent करना होगा ताकि हम इस बात के लिए Ensure हो सकें, कि हम जिस Entity के Data को प्राप्त करना चाहते हैं, हमें उसी Entity के Data प्राप्त होंगे।

उदाहरण के लिए मानलो कि Music Store के Database में Krishna नाम के दो Customers हैं। अब यदि Krishna नाम के Customer के Order की जानकारी प्राप्त करने के लिए Music Store में Searching की जाए, तो Music Store Application किस Krishna के Data Return करेगा? चूंकि दोनों Customers के नाम समान हैं, इसलिए Music Store Application दोनों ही Customers के Orders की List को Display करेगा।

क्योंकि हमारे इस Application में ऐसा कोई तरीका Use नहीं किया गया है, जिससे Music Store Application वांछित Customer के Orders की ही List Display करे। इस स्थिति में Music Store द्वारा Return किया जाने वाला Resultant Output Inaccurate होगा।

Music Store Application में इस समस्या के समाधान के रूप में हर Customer को एक Unique Customer Number प्रदान किया गया है और जब भी किसी Customer के Orders की जानकारी प्राप्त करनी होती है, तब उस Customer के नाम के स्थान पर उसके Customer Number का प्रयोग किया जाता है। Entities के Groups में से किसी Particular Entity को Identify करने का ये एक बहुत ही Common तरीका है, क्योंकि किसी भी Database में दो Customers को एक ही Customer Number प्रदान नहीं किया जाता है।

Particular Entity Instance को Identify करने के लिए हम एक दूसरा तरीका भी Use कर सकते हैं, जिसमें किसी Customer के First Name व Last Name को उसके Telephone Number के साथ परिभाषित कर सकते हैं। किसी Entity के इन Attributes के Combination द्वारा भी Customer को Uniquely Identify किया जा सकता है।

लेकिन इस तरीके में भी दो समस्याएं हैं। पहली ये कि जब Identifier बडा व Tricky होता है, तब इसके किसी भी हिस्से को Database में Enter करते समय Mistakes हो सकती हैं। दूसरी समस्या ये है कि किसी भी Customer का Phone Number Change हो सकता है, जिससे उस Customer का Identifier भी Change करना होगा और इस स्थिति में एक ही Customer के दो Identifier हो जाएंगे तथा एक ही Customer के दो Identifier होने की स्थिति में पैदा होने वाली समस्याओं के बारे में हम पहले ही पढ चुके हैं।

कुछ Entities जैसे कि Invoices आदि हमेंशा Natural Identifiers से Represent होते हैं, जिसे Invoice Number कहा जाता है। इस Invoice में Invoice Number का कोई विशेष अर्थ नहीं होता है, लेकिन फिर भी इस Invoice Number द्वारा किसी भी Invoice को Uniquely Identify किया जाता है। ठीक इसी तरह से हम किसी भी Entity को Uniquely Identify करने के लिए एक Meaningless Number का प्रयोग कर सकते हैं।

उदाहरण के लिए किसी Customer को Identify करने के लिए हमें उसके किसी Attribute या Attribute के Combination को Use करने की जरूरत नहीं है। हम किसी Customer को एक Meaningless Number द्वारा भी Identify कर सकते हैं और जब हम ऐसा करते हैं, तब Customer चाहे कहीं भी रहे, उसके किसी भी Attribute में चाहे जो Changes आए, उस Customer का Identifier Change नहीं होता और किसी Customer का Identifier Change होने की स्थिति में पैदा होने वाली परेशानियां भी Generate नहीं होती है।

हम जितनी बार भी किसी Entity के एक Instance को Database में Store करते हैं, हम यही चाहते हैं कि DBMS इस बात को Ensure करे कि हर नए Instance का एक Unique Identifier होगा। ये Concept Database **Constraint** का एक उदाहरण है।

Constraint एक ऐसा नियम या Rule होता है, जिसे Database Follow करता है। Database में विभिन्न प्रकार के Constraints को लागू कर देने पर Database उन Constraints या नियमों का पालन करता है, जिससे Database में Data के **Accurately** व **Consistently** Store होने की Guarantee हो जाती है।

Single-Valued and Multi-Valued Attributes

चूंकि हम एक Relational Database Design कर रहे हैं, इसलिए हमारे Data Model में हर Attribute Single-Valued होना जरूरी होता है। इसका मतलब ये हुआ कि किसी Entity के किसी Instance के हर Attribute में केवल एक ही मान (Data) को Store किया जा सकता है।

उदाहरण के लिए कोई Customer Entity अपने किसी भी Instance को केवल एक Telephone Number Store करने की सुविधा देता है। यानी हम किसी भी Customer का केवल एक ही Telephone Number Customer Table में Store कर सकते हैं।

लेकिन किसी Customer के पास एक से ज्यादा Telephone हो सकते हैं। यदि एक Customer के पास एक से ज्यादा Phone हैं और वह उन सभी Numbers को Database में Include करवाना चाहता है, तो Customer Entity यानी Customer Table में किसी एक Telephone Attribute File में एक से ज्यादा Telephone Numbers को Store नहीं किया जा सकता है। इस स्थिति को Customer Entity Handle नहीं कर सकता है।

हालांकि किसी Database का Entity-Relationship Model Database को Represent करने वाले Formal या Logical Data Model से अलग या Independent होता है, फिर भी हम Data के Entity-Relationship Model को Data के Logical Model के आधार पर ही Develop करते हैं।

चूंकि हम Logical Model में किसी Attribute में Multi-Valued मान को Store नहीं करते हैं, इसीलिए हम Entity-Relationship Create करते समय भी किसी Single Attribute Field में एक से ज्यादा Data या मानों को Store नहीं करते हैं।

एक से ज्यादा Phone Numbers की उपस्थिति Customer के Table या Customer Entity के Phone Number Attribute को एक Multi-Valued Attribute के रूप में परिभाषित कर देता है। क्योंकि किसी Relational Database में किसी भी Attribute Multi-Valued नहीं होता है, इसलिए हमें इन Multivalued Attributes को एक नया Entity Create करके Handle करना होता है और विभिन्न Multi-Valued मानों को उस नए Entity में Hold करना होता है।

एक से ज्यादा Phone Number होने की स्थिति में हम Phone Number नाम का एक Entity Create कर सकते हैं। इस Entity के हर Instance में एक Attribute उस Customer Number का होगा, जिसका Phone Number Store किया जाना है और दूसरा Attribute उस Customer के Phone Numbers का होगा।

यदि किसी Customer के चार Phone Numbers हों, तो इस Entity में उस Customer के चार Instance होंगे, जिनमें Customer Number तो समान होगा लेकिन Phone Numbers अलग–अलग होंगे। इस Concept को हम निम्नानुसार Represent कर सकते हैं, जहां पहला Entity एक Customer Entity है जिसमें चार Customer Instance हैं जबकि दूसरा Entity एक Phone Number Entity है, जिसमें हर Customer के एक से ज्यादा Phone Numbers उसके Customer Number के साथ Stored हैं।

Customer Entity

CustID	FName	LName	City	Dist.	State
001	Rahul	Sharma	Falna	Pali	Rajasthan
002	Rohit	Verma	Bali	Pali	Rajasthan
003	Krishna	Gopal	Beawer	Ajmer	Rajasthan
004	Madhav	Singh	Bhyender	Thana	Maharastra

Telephone Entity

CustID	PhoneNumber
001	9896589360
001	02934223366
001	02934223654
002	02938222333

```
   003            02937236598
   003            9979455505
   004            9357268933
//=================================================================
```

Telephone Number Entity में Telephone Numbers को Entity Identifier के रूप में Use किया जा सकता है। इससे Database में किसी प्रकार की कोई परेशानी पैदा नहीं होती है। क्योंकि इस Entity में हम केवल Phone Numbers को ही Store कर रहे हैं और एक Phone Number अपने आप में Unique होता है। जो Phone Number किसी Customer Number **001** के पास है वही Phone Number किसी Customer Number **003** के पास नहीं हो सकता।

Multi-Valued Attributes के साथ समस्या ये होती है कि यदि हम किसी Entity में Multi-Valued Attributes को Store करना चाहें, तो हमें हर Value के लिए एक नया Field Create करना होगा। यदि हम नए Fields Create कर भी लेते हैं, तब भी ये कभी भी निश्चित नहीं किया जा सकता कि हमें कुल कितने नए Fields Create करने चाहिए। क्योंकि किसी Customer के पास कितने Phone Numbers हो सकते हैं, हम इस बात को कभी भी निश्चित नहीं कर सकते हैं।

उदाहरण के लिए यदि हम हमारे Database में किसी Customer के अधिकतम 4 Phone Numbers Store कर सकने की सुविधा प्राप्त करने के लिए निम्नानुसार Entity को Design करते हैं:

Customer Entity

```
//=================================================================
CustID    FName     LName     ...    Phone01   Phone02   Phone03   Phone04
//=================================================================
001       Rahul     Sharma    ...    223355    445566    442255
002       Rohit     Verma     ...    121245    235689    214565    568996
003       Krishna   Gopal     ...    556688
004       Madhav    Singh     ...    558899    445566
//=================================================================
```

इस Entity Representation में हम देख सकते हैं कि केवल Customer Number 002 ही ऐसा Customer है, जिसके पास चार Phone हैं और केवल इसी Customer द्वारा Phone Numbers के लिए Reserve किए गए Storage Space का उपयोग किया जा रहा है। शेष Customers के पास चूंकि चार Phone नहीं हैं, इसलिए उनके इन Attributes द्वारा Reserve किए गए Space का कोई उपयोग नहीं हो रहा है।

यदि हम ये मान लें कि Customer Number **001** एक और Phone ले लेता है, तो उस पांचवे Phone Number को Store करने के लिए Entity के Database Representation को यानी Entity की Table को Modify करके एक और नया Field Customer Table में Add करना होगा, जो कि एक बहुत ही जटिल काम होता है और किसी समस्या को Solve करने के लिए हमेंशा

Database के Structure को Change करना भी सम्भव नहीं होता है, क्योंकि ऐसा करने पर Database के साथ Connected **Front-End** Forms को भी पूरी तरह से Change करना पडता है। यानी हम इस तरीके को तो किसी भी तरह से Apply नहीं कर सकते हैं।

जबकि हम समझ सकते हैं कि ज्यादातर लोगों के पास एक या दो ही Phone होते हैं, इस स्थिति में यदि हम इस तरीके को Use करते भी हैं, तब भी ज्यादातर Customers के पास केवल एक या दो Phone ही होने की वजह से इन Attributes के लिए Reserved Space का कोई उपयोग नहीं होता और ज्यादातर Space बिना मतलब के ही Reserved रहता है।

ऐसा करने पर Database की Size भी बढ जाती है, जिससे Database पर Searching Operations भी काफी समय लेता है। यानी किसी एक Entity में ही उसके किसी Multi-Valued मान को Store करने के लिए हम उसी Entity को Use नहीं कर सकते हैं। यदि हम ऐसा करते हैं, तो हम विभिन्न प्रकार की नई समस्याओं में फंस जाते हैं।

हालांकि Theoretically ये सम्भव है कि हम एक ऐसा Database Create कर सकते हैं, जो किसी एक Attribute में बहुत सारे Data Store कर सकता है, लेकिन Practically इस प्रकार के Database को Implement करना काफी मुश्किल होता है।

साथ ही यदि किसी Database के एक ही Field में एक से ज्यादा Data या मानों को Store किया जाता है, तो उस Database में Searching Operation केवल Sequential Form में ही हो सकता है, जो कि सबसे Slowest Searching Process होता है।

जबकि यदि एक Field में केवल एक ही Single Value या Data को Store किया जाता है, तो हम उस Database पर Binary Searching की Process को Apply कर सकते हैं, जो कि एक बहुत ही Fast Searching Process होता है।

एक सामान्य नियम के रूप में हम जब भी कभी किसी Multi-valued Attribute को Face कर रहे होते हैं, तो वह Attribute इस बात का संकेत होता है कि हमें Entity में और नए Fields Created करने पड सकते हैं। इसलिए Same Attribute के Multiple Values को Handle करने का सबसे सरल तरीका यही है कि हम एक नया Entity Create करें और Same Attribute के उन सभी Values को एक Instance की तरह उस Entity में Store करें।

Identifying Problem Related Entities

जिन समस्या से सम्बंधित वास्तविक Entity को Identify करना कई बार काफी Confusing होता है। यदि हम हमारे Music Store के ही उदाहरण के आधार पर समझें, तो क्या हम Music Store को Entity के रूप में Identify कर सकते हैं? नहीं, हम ऐसा नहीं कर सकते।

क्योंकि Music Store तो उन Entities का एक पूरा Collection है, जिन्हें Music Store Handle करता है। Entity तो वास्तव में Music Store के वे Items हैं, जिन्हें Database द्वारा Manage करना है।

इस बात को ठीक से समझने के लिए हम एक उदाहरण लेते हैं। मानलो कि हम Music Store को ही एक Entity के रूप में Identify कर लेते हैं। अब इस Entity को Represent करने के लिए हमें इसके विभिन्न Attributes का पता करना होगा जो कि Music Store के Items Numbers, Item Titles, Item In Stock, Retail Price आदि होंगे।

लेकिन चूंकि हम पूरे Music Store को ही एक Single Entity के रूप में Describe कर रहे हैं, इसलिए हमें इसके हर Attribute में Multiple Values को Store करना होगा और जैसा कि हमने पहले बताया कि किसी भी Relational Database में कोई भी Attribute Multi-Valued नहीं हो सकता, इसलिए हम Music Store को एक Single Entity के रूप में Identify नहीं कर सकते हैं, बल्कि हमें इसे Entities के एक Collection के रूप में Identify करना होता है।

एक और उदाहरण देखते हैं, मानलो कि कोई Doctor अपने सभी Patient की Medical History को Maintain करता है। किसी Music Store के Inventory Program की तरह ही Medical History भी एक तरह का एक से ज्यादा Entities का Collection है। एक Medical History Appointments व उन Appointments के दौरान होने वाली घटनाओं द्वारा बनता है।

इसलिए ये History वास्तव में Appointment Entities व Medical Treatment Entities के Instances का Collection है, ना कि स्वयं एक Entity है। "**History**" तो वह Output है जिसे एक Database Application उसके Entities के Instances (Records) के Collection के रूप में Generate करता है।

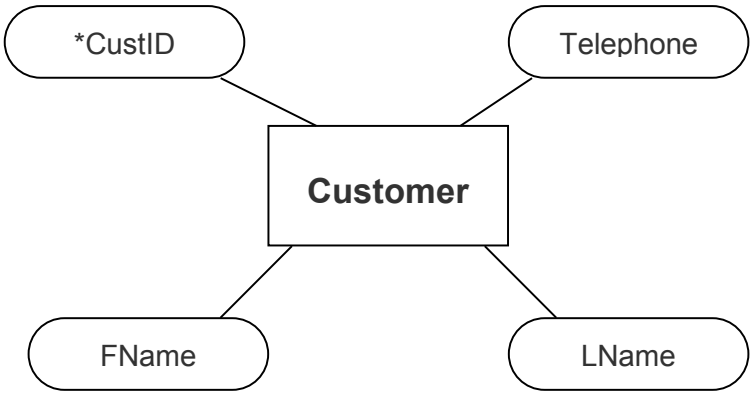
Documenting Logical Data Relationships

Entity-Relationship Diagram हमें एक ऐसा तरीका Provide करता है, जिसका प्रयोग करके हम किसी Entity को उसे Describe करने वाले Attributes के साथ Logically Represent कर सकते हैं। ER Diagrams के कई तरीके प्रचलित हैं, लेकिन दो तरीकों को सबसे ज्यादा Use किया जाता है।

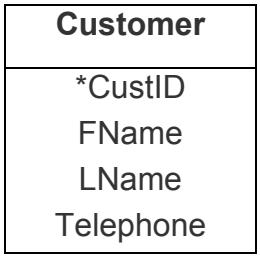
पहला तरीका Dr. Peter P. S. Chen ने Develop किया था। इसलिए इस Data Modeling Diagram को ER Chen नाम दिया गया है। दूसरा तरीका James Martin व Clive Finkelstein ने Develop किया है और इसे Information Engineering (IE) में Use किया जाता है।

हम इन में से जिन भी तरीके को Use करके अपने Database का Entity Relationship Data Model Diagram बना सकते हैं। दोनों ही तरीकों में जिन Entity को एक Rectangle द्वारा Represent किया जाता है और हर Entity का नाम उसके Rectangle के अन्दर लिखा जाता है। उदाहरण के लिए हम Customer Entity को ER Diagram में निम्नानुसार Represent कर सकते हैं:

Original Chen के ER Diagram Model में Entities के साथ उनके Attributes को Show करने की कोई सुविधा नहीं थी। फिर भी लोगों ने Chen के इस ER Model को थोडा सा Modify करके निम्नानुसार Entity के Attributes को भी Entity के साथ Include कर लिया है।



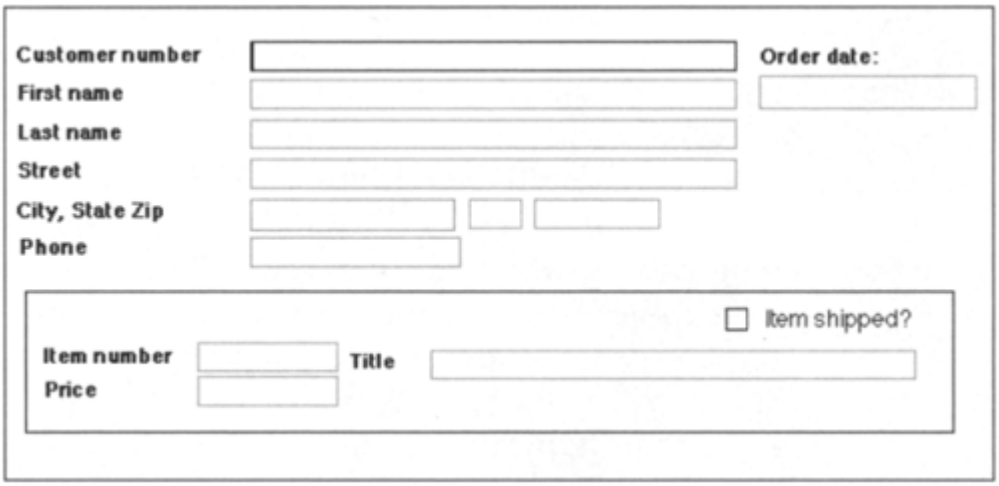
Entity के Identifier Attribute के नाम के पहले एक Asterisk (*) लगाकर Identifier Attribute को Represent किया जाता है। Information Engineering Model में Entity को उसके Attributes के साथ में निम्नानुसार चित्र द्वारा Represent किया जाता है:



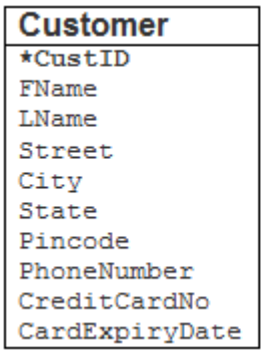
चूंकि, Information Engineering Model को कम Space में ज्यादा अच्छे तरीके से बनाया जा सकता है, इसलिए हम इस पुस्तक में ज्यादातर इसी Model के Symbols को Use करेंगे। हालांकि जरूरत होने पर Chen के **ER Model** को भी Use किया जाएगा।

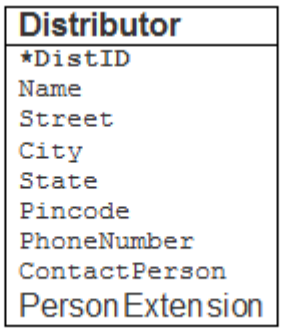
Entities and Attributes for Music Store Application

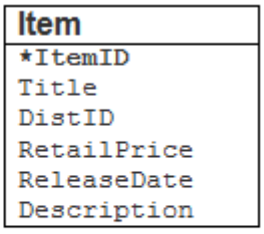
हमारे Music Store Database के Order Entry Form से ही हमें उन मुख्य Entities का पता चल जाता है, जिन्हें हमें Music Store Database में Manage करना है।

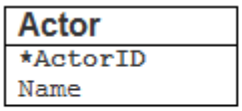


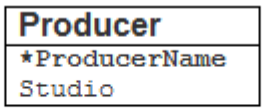
हालांकि जब हम Database Design Process को आगे बढाते हैं, तब Database Design में अन्य Additional Entities की भी जरूरत पडती है। हमारे Music Store Database से सम्बंधित मुख्य–मुख्य Entities Customer, Order, Distributor, Actor, Producer व Item हैं और इन Entities से सम्बंधित वे Attributes जिन्हें Database में Store करना हैं, उन्हें निम्नानुसार Information Engineering Diagram द्वारा दर्शाया गया है:

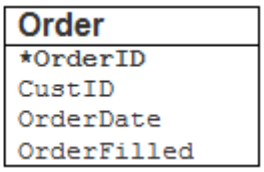












Domains

हर Attribute का एक Domain होता है, जो ये Express करता है कि Particular जिन Attribute में किस तरह के मान Store हो सकेंगे। एक Domain बहुत छोटा भी हो सकता है।

उदाहरण के लिए जिन Population से सम्बंधित Database में **Male/Female** को Represent करने के लिए **Gender** Attribute में केवल **M/F** Character को ही Store करने की जरूरत होती है। जबकि जिन Customer का नाम Store करने के लिए Name Attribute में बहुत सारे Characters को Store करना पडता है।

एक Database Management System (DBMS) एक Domain Constraint के Through एक Domain Enforce करता है। उसके बाद जब भी Database में कोई Data या मान Store किया जाता है, वह DBMS Software उस Domain Constraint के आधार पर ये तय करता है कि Database में जाने वाला मान उसी Domain का है जिसके लिए उसे Domain Constraint द्वारा Set किया गया है अथवा नहीं।

उदाहरण के लिए यदि जिन Customer की Birth Date को Store करने के लिए Database में DOB Attribute को **Date/Time** Domain Constraint से को Set किया गया है, तो DBMS Software उस Attribute में केवल उसी Data को Store करेगा जो Date/Time Format का होगा।

शेष जिन भी अन्य Domain (Data Type) के Data को वह DBMS Software Database में Store नहीं करने देगा यहां तक कि जब हम **Date/Time** Domain Constraint को जिन Attribute के साथ Set करते हैं, तब हम उस Attribute में **30 February** जैसी जिन गलत Date को भी Store नहीं कर सकते हैं।

Documenting Domains

ER Diagram का जो Common Format Use किया जाता है, उसमें Domain को Specify करने की कोई सुविधा नहीं होती है, बल्कि उस Diagram से एक Document Associated रहता है, जिसे Data Dictionary कहते हैं। Data Dictionary के बारे में हम आगे विस्तार से पढेंगे। हम Chen के ER Model में हर Attribute के नीचे उस Attribute के Domain को Specify कर सकते हैं।

Practical Domain Choices

Music Store के Entities के Attributes के लिए जिन Domains को Choose किया जाता है, वे Theoretically उन DBMS Softwares से Independent होने चाहिए, जिन्हें हम Music Store Database को Develop करने के लिए Use करेंगे।

फिर भी ज्यादातर Relational DBMS अपनी Query Language के रूप में SQL का प्रयोग करते हैं, जिसमें जिन Attribute को निम्न Domain Assign किए जा सकते हैं:

CHAR

ये एक Fixed-Length Domain होता है। इस Domain को जिन Attribute के साथ Associate करने पर हम उस Attribute में अधिकतम **256** Characters Store कर सकते हैं।

VARCHAR

ये एक Variable-Length Domain होता है। इस Domain को जिन Attribute के साथ Associate करने पर हम उस Attribute में Variable Length के अधिकतम **256** Characters Store कर सकते हैं।

INT

जब हमें जिन Attribute में Integer मान Store करना होता है, तब हम उस Attribute के साथ इस Domain को Associate करते हैं।

DECIMAL and NUMERIC

जब हमें जिन Attribute में दसमलव वाले मान को Store करना होता है, तब हम उस Attribute के साथ इस Domain को Associate करते हैं। जब हम एक Real Number Domain को जिन Attribute के साथ Associated करते हैं, तब हमें ये भी Specify करना होता है कि हम दसमलव के बाद के अंकों सहित कितने अंकों तक का मान Store करना चाहते हैं और दसमलव के बाद कुल कितने अंकों तक का मान Store करना चाहते हैं।

उदाहरण के लिए Currency Values को हमेंशा दसमलव के बाद दो संख्याओं तक Specify करना होता है, इसलिए यदि हम Currency को Computer में Store करने के लिए जिन Attribute के साथ इस Domain को Associate करना चाहें, तो हम DECIMAL (6, 2) Statement द्वारा ये काम कर सकते हैं, जहां कुल 6 अंकों तक के Currency मान को जिन Attribute में Store किया जा सकता है जबकि उस मान में दसमलव के बाद दो संख्याओं तक को Store किया जा सकता है।

DATE

जब हमें जिन Attribute में **Date** Store करना होता है, तब हम उस Attribute के साथ इस Domain को Associate करते हैं।

TIME

जब हमें जिन Attribute में **Time** Store करना होता है, तब हम उस Attribute के साथ इस Domain को Associate करते हैं।

DATETIME

जब हमें जिन Attribute में **Date** व **Time** के Combination को एक साथ Store करना होता है, तब हम उस Attribute के साथ इस Domain को Associate करते हैं।

BOOLEAN

जब हमें जिन Attribute में **True** या **False** जैसी जिन Logical Value को Store करना होता है, तब हम उस Attribute के साथ इस Domain को Associate करते हैं।

आज के नए DBMS Softwares BLOB नाम के एक नए Data Type को भी Support करते हैं, जिसका प्रयोग बडे Binary Object जैसे कि जिन Graphical Image को Store करने के लिए कर सकते हैं।

जिन Attribute के लिए एक उचित Domain को Choose करने पर Database की Accuracy को पूरी तरह से सुनिश्चित किया जा सकता है।

उदाहरण के लिए हमारे देश में Pincode Number 6 Digit का एक Number होता है। क्या एक Pincode Number को INT Domain से Associate करना चाहिए। नहीं, एक Pincode को दो कारणों की वजह से INT Domain से Associate नहीं करना चाहिए।

पहला कारण ये है कि कई देशों में Pincode Numbers के बीच में Hyphen Symbol का प्रयोग किया जाता है और दूसरा कारण ये है कि यदि Pincode Number को INT Domain के साथ Store किया जाएगा, तो कई Pincode Number के पहले लगाया जाने वाला Zero नहीं लगाया जा सकेगा, क्योंकि DBMS उस Preceding Zero को Remove कर देगा।

साथ ही Pincode एक ऐसा Number होता है, जिसके साथ हम कभी भी इस प्रकार की Arithmetical Calculations Perform नहीं करते हैं, इसलिए Pincode Number को Integer के बजाय Character Form में Store करने पर हमें इस प्रकार की परेशानी का सामना नहीं करना पडेगा।

इसी तरह से जब हमें जिस Data को साथ जिस प्रकार की Calculation करने की जरूरत होती है, तब हम उस Data को Character Form में Store नहीं कर सकते हैं।

उदाहरण के लिए यदि हम जिन Employee की Salary को यदि Character Form में Store किया जाए, तो हम उस Employee का PF, DA, HRA आदि Calculate नहीं कर सकते हैं, क्योंकि Character Domain पर जिस प्रकार का Arithmetical Operation Perform नहीं किया जा सकता।

इसी तरह से यदि हम जिन Attribute में Date या Time Store करना चाहें, तो हमें DATE या TIME Domains को ही Use करना चाहिए। यदि हम जिस Date को Store करने के लिए Character Format का प्रयोग करते हैं, तो हम Date से सम्बंधित विभिन्न प्रकार की समस्याओं में फंस जाएंगे।

उदाहरण के लिए मानलो कि हम दो Dates 01/11/2009 व 10/11/2000 को Character Domain का प्रयोग करके Database में Store करते हैं। अब यदि हम DBMS से ये पता करना चाहें कि कौनसी Date पहले आती है तो दोनों ही Dates Character Format में Store होने के कारण DBMS इन दोनों Dates में से Alphabetical Order के आधार पर पहले आने वाली Date को Calculate करेगा और हमें Result के रूप में 01/11/2009 प्रदान करेगा, क्योंकि Alphabetical Order में 01, 10 से पहले आता है और हम समझ सकते हैं कि ये एक गलत Result है।

यदि हम Character Form में ही Date को Store करके ये जानना चाहें कि कौनसी Date पहले आती है, तो हमें इस Date को **YYYY/MM/DD** Format में Database में Store करना होगा और इस Format की Date को सारी दुनियां में बहुत ही Rarely Use किया जाता है। जबकि यदि हम इन Dates को Store करने के लिए **DATE** Domain का प्रयोग करते हैं, तो हमें Date से सम्बंधित इस प्रकार की जिन समस्या का सामना नहीं करना पडता है, साथ ही हम DBMS द्वारा Date के लिए Provide की जाने वाली सुविधाओं को भी प्राप्त कर सकते हैं। यानी हम दो Dates के बीच Difference का पता लगा सकते हैं या दो Dates को आपस में Compare कर सकते हैं।

DBMS

BASIC DATA RELATIONSHIP

DBMS – Basic Data Relationships

एक बार अपने Database से सम्बंधित सभी Basic Entities का पता लग जाने के बाद हमारा अगला काम उन Entities के बीच की आपसी Relationship को Identify करना होता है। किसी Database के विभिन्न Entities के बीच मुख्यतः तीन तरह की Relationship हो सकती हैं: One To One Relationship, One To Many Relationship व Many To Many Relationship

इससे पहले कि हम इन Relationships को समझें हमें एक बात ध्यान में रखनी होती है कि किसी Database में जितनी भी Relationships Stored होती हैं, वे सभी Relationships Entities के Instances के बीच होती हैं।

उदाहरण के लिए हमारे Music Store Example के आधार पर कोई Customer उन Items से Related होता है, जिनका उस Customer ने Order दिया है। यानी Customer Entity का हर Instance Item Entity के Order किए गए Item Instance से Related होता है।

हम यहां जिस Relationship के बारे में चर्चा कर रहे हैं, वह Relationship पूरी तरह से किसी Database का Conceptual Representation है और इस Relationship का Data के Actual Physical Storage से कोई सम्बंध नहीं होता है।

जब हम Data Relationship को ER Diagram या IE Diagram द्वारा Document करते हैं, तब हम विभिन्न Entities के बीच की Relationship को Show करते हैं। इन Diagrams में हम उन Possible Relationships को Show करते हैं जो Database में Allowable होती हैं। हम जब तक किसी Relationship को Compulsory रूप से Specify नहीं करते हैं, तब तक ये जरूरी नहीं होता है कि Database से Related हर Entity आपसी Relationship में Involved हो।

उदाहरण के लिए Music Store Database किसी भी Customer की Information को Database में Store कर सकता है, भले ही उस Customer ने Music Store को किसी भी Item के लिए कोई भी Order ना दिया हो। यहां ये जरूरी नहीं है कि वही Customer Music Store Database में Store हो सकता है, जिसने कोई Music Store को किसी Title के लिए कोई Order दिया हो।

One To One Relationships

मानलो कि किसी छोटे से शहर में केवल एक ही Airport है और किसी Database में वह शहर व Airport दोनों ही एक Entity के रूप में Involved हैं। इस स्थिति में City व Airport दोनों ही उस Database में दो अलग Entities के Instance के रूप में Represent होंगे।

अतः City व Airport के बीच **One To One** की Relationship को इस तरह से Express किया जा सकता है कि Airport केवल एक City में ही Situated है और उस City में केवल एक ही Airport है।

ये उदाहरण **One To One** Relationship का एक आदर्श उदाहरण है, क्योंकि इस Database में किसी भी समय एक City से केवल एक ही Airport Related होगा और उस एक Airport से केवल एक ही City Related होगी। हालांकि किसी शहर में एक से ज्यादा Airports हो सकते हैं, लेकिन हम यहां एक बहुत ही छोटे शहर के Airport Database को Manage कर रहे हैं और हम ये मान रहे हैं कि किसी भी छोटे शहर में एक से ज्यादा Airport नहीं हो सकते हैं।

यदि हमारे पास दो Entities **A** व **B** के दो Instances **Ai** व **Bi** हों, तो इन दोनों Entities के Instances के बीच ***One To One*** की Relationship केवल तभी सम्भव है, जब किसी भी समय **Ai** Entity **B** के **Zero** या **One** Instance से Related हो और **Bi** Entity **A** के **Zero** या **One** Instance से Related हो।

किसी भी Business Database Application में ***One To One*** की Relationship बहुत ही Rare Case में बनती है। उदाहरण के लिए मानलो कि हमारा Music Store एक नए Distributor से Dealing करना तय करता है। Music Store उस नए Distributor को केवल एक ही Special Title का Order देता है।

अब यदि हम उस नए Distributor को अपने Database में स्थान दें, तो हम देखते हैं कि उस नए **Distributor Entity** का Instance **Music Store** के **Item Entity** के केवल एक Instance से Related होता है। इसलिए यहां पर उस नए Distributor व Item के बीच One To One की Relationship बनती है।

अब यदि Music Store उस नए Distributor को कई और Titles के Order देता है, तो यहां ये नियम Violate हो जाता है कि वह नया Distributor Music Store के केवल एक ही Item से Related है। इस स्थिति में उस नए Distributor Entity के Instance व Item Entity के Instance के बीच One To One के स्थान पर One To Many की Relationship बन जाती है। क्योंकि अब वह नया Distributor Music Store के एक से ज्यादा Items के Titles से Related हो जाता है।

इसी तरह से यदि हम Music Store Database में Credit Card नाम का एक और Entity Create करते हैं, जिसमें उन Credit Cards के Data को Hold किया जाता है, जिनका प्रयोग वे Customer अपनी उधारी चुकाने के लिए करते हैं, जिनका कुछ पैसा Music Store में बकाया है। चूंकि हर Customer केवल एक ही Credit Card से अपना पैसा चुकाता है, इसलिए यहां पर भी Customer Entity के Instance व Credit Card Entity के Instance के बीच ***One To One*** की Relationship Create होती है।

Credit Card Entity के लिए Credit Card का Number, Type व Expiration Date Attributes हो सकते हैं। यदि हम ये मान लें कि हर Customer के पास केवल एक ही Credit Card होता है, तो चूंकि Credit Card के किसी भी Attribute का मान Multi-Valued ना होने की वजह से हमें Credit Card को एक अलग Entity के रूप में Represent करने की जरूरत नहीं है। हम Credit Card के विभिन्न Attributes को Customer की Information के साथ Store कर सकते हैं।

हम जब भी कभी किसी Database के सन्दर्भ में ***One To One*** Relationship को Identify करते हैं, तो हमें ध्यान से ये देख लेना चाहिए, कि कहीं हम ***One To Many*** की Relationship को तो ***One To One*** की Relationship नहीं मान रहे हैं या कहीं एक ही Entity को तो दो Entity के रूप में Represent करने की कोशिश तो नहीं कर रहे हैं।

One To Many Relationships

किसी Database के विभिन्न Entities के बीच की ये एक बहुत ही Common Relationship होती है। वास्तव में Maximum Database में जितनी भी Relationships होती हैं, उनमें से ज्यादातर ***One To Many*** की ही Relationship होती हैं। One To One की Relationship तो Rare Case में ही बनती है।

Music Store Database के सन्दर्भ में भी Database के विभिन्न Entities के Instances के बीच ये ही Relationship Define हो रही है। उदाहरण के लिए Music Store बहुत सारे Titles के लिए Distributors को Order दे सकता है और Music Store एक Title का Item केवल एक ही Distributors से प्राप्त करता है।

इसी तरह से एक Customer Music Store पर कई Orders Place कर सकता है लेकिन एक Order केवल एक ही Customer देता है। जैसे Order Number **001** किसी Rahul नाम के Customer ने दिया हो, तो इसी Number का कोई दूसरा Order किसी दूसरे Customer द्वारा Music Store को नहीं दिया जा सकता है।

यदि हमारे पास दो Entities **A** व **B** के दो Instances **Ai** व **Bi** हों, तो इन दोनों Entities के Instances के बीच ***One To Many*** की Relationship केवल तभी सम्भव है, जब किसी भी समय **Ai** Entity **B** के **Zero, One** या **More** Instances से Related हो और **Bi** Entity **A** के **Zero** या **One** Instance से Related हो।

One To Many की Relationship को एक Family Relationship के रूप में आदर्श तरीके से Represent किया जा सकता है। एक मां व उसके बच्चों के बीच ***One To Many*** की Relationship होती है, जिसमें एक मां के कई बच्चे तो हो सकते हैं, लेकिन किसी भी बच्चे की केवल एक ही मां हो सकती है।

ठीक इसी तरह से यदि हम एक और उदाहरण देखें तो एक Computer व उसके CPU के बीच भी ***One To Many*** की Relationship को Represent किया जा सकता है, जहां एक CPU को केवल एक ही Computer में Install किया जा सकता, जबकि एक Computer में एक से ज्यादा CPU को Install किया जा सकता है।

यदि इसी Concept को हम Music Store के Database पर Apply करें, तो हम कह सकते हैं कि Music Store व उसके Distributor के बीच ***One To Many*** की Relationship होती है, जहां Music Store एक Title के लिए किसी एक ही Distributor को Order दे सकता है, जबकि वही Title **Music Store** को एक से ज्यादा Distributor प्रदान कर सकते हैं।

एक बात ध्यान रखें कि जब हम Data Relationships को Specify कर रहे होते हैं, तब हम सभी सम्भव Relationships को Indicate कर रहे होते हैं और ये जरूरी नहीं होता है कि सभी Entities के सभी Instances हर Documented Relationship में भाग लें। यानी हमारे Music Store Database के आधार पर ये जरूरी नहीं है कि कोई Distributor Instance किसी Item Entity के Zero, एक या एक से ज्यादा Instances से Related हो।

Many To Many Relationships

ये Relationships भी काफी Common Relationships हैं। यदि हम Music Store Database के सन्दर्भ में इस Relationship को परिभाषित करें, तो Customer Entity व Item Entity के बीच ***Many To Many*** की Relationship होती है। क्योंकि एक Customer एक से ज्यादा Titles का Order दे सकता है और एक ही Title के लिए एक से ज्यादा Customer Order दे सकते हैं।

इसी तरह से एक Distributor व एक Item के बीच भी ***Many To Many*** की Relationship हो सकती है, जिसमें एक ही Distributor को एक से ज्यादा Title के लिए Order दिया जा सकता है और एक ही Title को एक से ज्यादा Orders में Place किया जा सकता है।

यदि हमारे पास दो Entities **A** व **B** के दो Instances **Ai** व **Bi** हों, तो इन दोनों Entities के Instances के बीच ***Many To Many*** की Relationship केवल तभी सम्भव है, जब किसी भी समय **Ai** Entity **B** के **Zero, One** या **More** Instances से Related और **Bi** Entity **A** के **Zero, One** या **More** Instances से Related. Many To Many Relationship Database Design में दो बडी समस्याएं पैदा करता है, जिन्हें आगे विस्तार से समझाया गया है।

Weak Entities and Mandatory Relationships

हमने विभिन्न Relationships को Discuss करते समय Relationship को "**Zero**" Instance से भी Related बताया है, जो इस बात को Indicate करता है कि आपसी Relationship में Specify किए गए Entities के किसी Instances का Relationship में Participate करना Optional है।

उदाहरण के लिए यदि हम **Music Store** Database के सन्दर्भ में इस बात को समझें, तो एक Customer की Information को उस समय भी Database में Store किया जा सकता है, जबकि उसने किसी Item के लिए कोई Order नहीं किया होता है।

इस स्थिति में एक Customer Entity का कोई Instance Item Entity के किसी भी Instance से Related नहीं होता है या दूसरे शब्दों में कहें तो एक Customer Entity का कोई Instance Item Entity के **Zero** Instance से Related होता है।

हालांकि एक Customer जब कोई Order Place नहीं करता है, तब भी उस Customer की Database में Entry की जा सकती है, लेकिन यदि हम इस Concept को Reverse Order में लें, तो ऐसा सम्भव नहीं हो सकता। यानी हर Order का किसी एक Customer से Related होना जरूरी होता है। बिना किसी Customer के Order दिए हुए, Music Store Database में एक Order Place ही नहीं हो सकता, क्योंकि किसी भी Order को Place करने के लिए एक Customer जरूर होना चाहिए।

यदि हम इस Discussion के आधार पर समझें तो यहां **Order** एक ***Weak Entity*** है, क्योंकि ये एक ऐसा Entity है, जिसे Database में तब तक Store नहीं किया जा सकता है, जब तक कि उस Order Instance से Related कोई दूसरा Entity Instance उस Database में Present ना हो और उस Order से Related ना हो।

Customer Entity का एक Instance Zero, One या More Orders से Related हो सकता है। फिर भी एक Order का केवल एक और सिर्फ एक ही Customer Entity Instance से Related होना जरूरी होता है। Weak Entity के लिए Zero Option Available नहीं होता है। इस स्थिति में **Customer** व **Order** के बीच की Relationship एक जरूरी यानी **Mandatory** या **Compulsory** Relationship है, जिसे Database को ठीक से Manage करने के लिए Database में परिभाषित करना जरूरी होता है।

किसी Database की **Consistency** व **Integrity** को Maintain करने के लिए उसके सभी Weak Entities व उस Weak Entity से Associated Mandatory Relationship Entity को Identify करना काफी महत्वपूर्ण होता है।

इनका Database पर आपसी प्रभाव जानने के लिए मानलो कि हम एक ऐसा Order Database में Store करते हैं, जिससे Belonged Customer का पता नहीं हैं। इस स्थिति में हम उस Order में

Specify किए गए Items को कभी भी किसी Customer तक Ship नहीं कर सकते हैं, क्योंकि उस Order में उस Customer की जानकारी ही नहीं होती है, जिसे उस Order के Items Ship करने हैं।

इसी Concept के अनुसार हम **Order** व **Order Lines** के बीच की Relationship को भी ***One To Many*** के Relationship के रूप में Define करते हैं, क्योंकि हम नहीं चाहते हैं कि बिना किसी Order से Related हुए कोई Order Line Database में Store हो। Order Line किसी Order पर स्थित किसी Specific Item को Refer करता है। यानी एक Order Line तब तक Meaningless होता है, जब तक कि हम ये नहीं जानते हैं कि वह Order Line किस Order से Belong करता है।

इसके बजाय हम Music Store पर उपलब्ध किसी भी Item को Database में Store कर सकते हैं, जबकि हमें ये जानने की जरूरत नहीं होती है कि उस Item को किस Supplier ने भेजा है, जबकि हम यहां पर ये मान रहे हैं कि हर Item किसी एक Supplier से आता है। इस स्थिति में Supplier व Item के बीच वास्तव में **Zero To Many** की Relationship हो जाती है।

Documenting Relationships

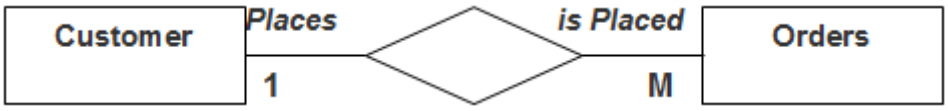
Chen व IE दोनों ही तरीकों के ER Diagrams में Relationship को अलग तरीकों से Represent किया जाता है। दोनो ही तरीकों के अपने फायदे व कमियां हैं। Chen Method में Relationship को Represent करने के लिए Diamond Symbol का प्रयोग किया जाता है और Entities के बीच Relationship के Type को Represent करने के लिए Arrow Based Lines का प्रयोग किया जाता है। उदाहरण के लिए निम्न ER Diagram को देखिए:

इस Diagram में हम **Customers** व **Orders** के बीच की Relationship को देख सकते हैं। Customer की तरफ जो Single Arrow Point कर रहा है वह Arrow Indicate करता है कि एक Order केवल एक ही Customer से Belong कर रहा है। जबकि Orders Entity की तरफ का Double Arrow इस बात को Indicate करता है कि एक Customer एक से ज्यादा Orders Place कर सकता है। Diamond के बीच लिखा गया शब्द Relationship के सम्बंध में कुछ जानकारी प्रदान करता है।

Chen Model में दो Alternative Styles हैं। पहले Style में Arrows को Numbers व Letters से Replace कर दिया जाता है। Number "**1**" ये Indication देता है कि एक Order किसी एक Customer से आता है। जबकि "**M**" या "**N**" Character ये Indication देता है कि Customer

एक से ज्यादा Orders Place कर सकता है। इस तरीके को Apply करने पर हमें निम्नानुसार Diagram प्राप्त होता है:

दूसरा Alternative उस Problem का Solution दर्शाता है, जो तब पैदा होती है जब हम Relationship को दोनों Directions में Read करते हैं। यानी "**Customer Places Order**" तो एक Meaningful Information है। लेकिन "**Order Places Customer**" किसी तरह की कोई Meaningful Information प्रदान नहीं करता है। इस समस्या को Solve करने के लिए **ER Diagram** में Diamond से Relationship Represent करने वाले Arrows व Relationship के नाम को Remove कर दिया जाता है और Diagram को निम्नानुसार Inverse कर लिया जाता है:



अब ये Diagram ज्यादा Meaningful Information Provide करता है, जो निम्नानुसार है कि :

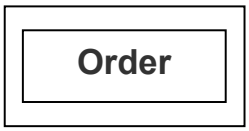
1 **1** Customer **M** Orders Place कर सकता है। यानी
"*1 Customer Places Many Orders*" और

2 **1** Order केवल **1** Customer ही Place कर सकता है। यानी
"1 *Order is Placed by* 1 *Customer*"

Chen के Model में एक बहुत ही बडी Limitation है और वो है ER Diagram को Draw करने की। Chen के ER Diagram में किसी Weak Entity व Mandatory Relationship को Represent करने का कोई स्पष्ट तरीका नहीं है।

उदाहरण के लिए Music Store Database के लिए Order एक Weak Entity है और बिना किसी Customer के ये Entity Database में Store नहीं किया जा सकता और Customer Entity के साथ इसकी Relationship जरूरी यानी Mandatory है।

कुछ Database Designers ने Chen Method में एक नया Symbol Add किया है जिसमें किसी Weak Entity को निम्नानुसार एक Double Bordered Rectangle द्वारा दर्शाया जाता है:



जब भी कभी किसी ER Diagram में किसी Weak Entity को Represent किया जाता है, तब ये Symbol इस बात को Indicate करता है कि ये Entity व इसके एक Parent Entity दोनों के बीच में **Mandatory Relationship** है। लेकिन यदि इस Entity के साथ एक से ज्यादा Parent Entities Related हों, तो इस बात को सामान्य तरीके से तय करना मुश्किल हो जाता है कि कौनसे Entity के साथ इस Weak Entity की **Relationship Mandatory** है।

Chen Method के अलावा हम Information Engineering Method को Use करके भी ER Relationship Model को Create कर सकते हैं। IE Method में Line के End के आधार पर ही विभिन्न प्रकार की Relationships को Represent किया जाता है, जो कि Chen के Method की तुलना में ER Diagram को ज्यादा सरल बना देता है।

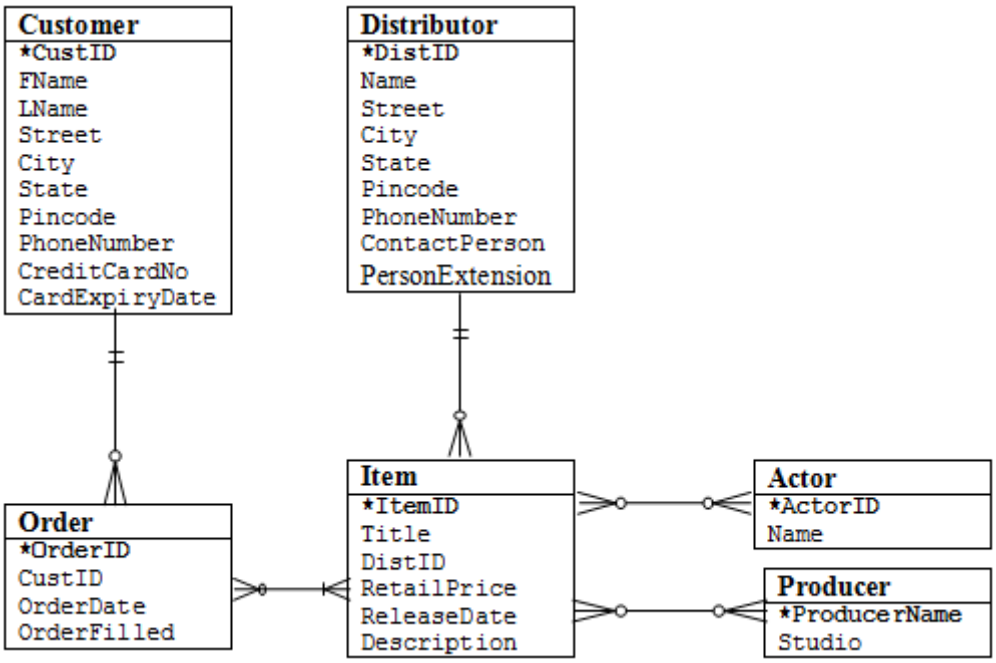
उदाहरण के लिए निम्न चित्र में उसी One To Many Relationship को दर्शाया गया है, जिसे हमने Chen के Method द्वारा दर्शाया है। इस चित्र में Line के End के आधार पर हम इस बात का पता लगा सकते हैं कि कौनसी Relationship Mandatory है और कौनसा Entity एक Weak Entity है।

Customer Entity के आगे जो दो खडी Lines (||) हैं, उनका मतलब ये होता है कि हर **Order** सिर्फ और सिर्फ एक **Customer** से Related है। चूंकि **0** Optional नहीं है, इसलिए इन Entities के बीच की Relationship **Mandatory** है।

इसी तरह से Order Entity के साथ Connect किए गए **0** व तीन Legs का मतलब ये है कि एक Customer **Zero, One** या **More** Orders से Related हो सकता है। एक IE Diagram में Line के अन्त में मुख्यतः चार Symbols को Use किया जा सकता है:

1 || : One and Only One (Mandatory Relationship)
2 0| : Zero or One
3 >| : One or More (Mandatory Relationship)
4 >0 : Zero, One or More

अब यदि हम Music Store Database के विभिन्न Entities के बीच की Relationships को IE Diagram द्वारा Represent करें, तो बनने वाला Diagram निम्नानुसार बनेगाः



इस IE Diagram द्वारा निम्न Relationships Model हो रहे हैंः

1 एक Customer Zero, One या More Orders Place कर सकता है जबकि एक Order एक और सिर्फ एक Customer द्वारा Place किया जा सकता है।

2 एक Order में एक या एक से ज्यादा Items हो सकते हैं जबकि एक Item Zero, One या More Orders में Appear हो सकता है।

3 एक Actor Zero, One या More Items में Appear हो सकता है और इसी तरह से एक Item में Zero, One या More Actors हो सकते हैं। (कई बार ऐसी Films भी बनती हैं, जिनमें Human Actors के स्थान पर Animal Actors होते हैं। इसलिए हमेंशा ये जरूरी नहीं होता है कि हर Item कम से कम एक Actor से Related हो।)

4 हर Item के Zero, One या More Producers हो सकते हैं और हर Producer Zero, One या More Items के लिए Responsible होता है। हालांकि हम Practically तब

तक किसी Producer को Database में Store नहीं करते हैं, जब तक कि वह Producer किसी Item से Related ना हो। Producer व Items के बीच की Relationship को Optional Means के रूप में Specify करके हम बिना Item की Information Store किए हुए भी Producer की Information को Database में Store कर सकते हैं।

इस Design में Notice करने वाली जो सबसे बडी चीज है वो ये है कि इस Design में तीन Many To Many Relationships हैं, जो कि Order To Item, Actor To Item व Producer To Item के Entity के बीच है। इससे पहले कि हम इस Data Model को किसी Relational Database पर Map करें, इन्हें किसी अन्य तरीके से Handle करना होगा, क्योंकि किसी भी Relational Database में हम Many To Many Relationship को Directly Map नहीं कर सकते हैं।

Dealing with Many To Many Relationships

जैसाकि हमने पहले बताया है कि ***Many To Many*** Relationship के साथ कुछ अलग प्रकार की समस्याएं हैं। सबसे पहली समस्या तो यही है कि कोई भी Relational Data Model **Many To Many** Relationship को Directly Handle नहीं कर सकता है। किसी Relational Data Model में हम केवल ***One To One*** या ***One To Many*** Relationship को ही Handle कर सकते हैं।

इसका मतलब ये हुआ कि हमने **Music Store** Database में जिस Many To Many Relationship को Identify किया है, उन्हें One To Many Relationships के Collections के रूप में Convert करके Relational Data Model में Use करना होगा, ताकि हम इन्हें एक Relational DBMS में Use कर सकें।

दूसरी समस्या थोडी ज्यादा जटिल है। इसे समझने के लिए मानलो कि Music Store किसी Distributor को कोई Order देता है और Music Store पर कोई Customer Order Place करता है। इसलिए Order व Item के बीच Many To Many की Relationship बनती है, क्योंकि हर Order में एक से ज्यादा Items Appear हो सकते हैं और बाद में हर Item कई Orders पर Appear हो सकता है। जब कभी Music Store किसी Item के लिए कोई Order Place करता है, तब Item के Copies की संख्या Music Store को प्राप्त होने वाली Copies की संख्या से भिन्न हो सकती है। यानी हो सकता है कि Music Store किसी Item के चार Copy प्राप्त करना चाहता हो जबकि Stock की कमी के कारण उसे वह Item केवल तीन ही प्राप्त हो।

अब सवाल ये है कि Order किए गए Item की Quantity को कहां Store करना चाहिए? क्योंकि ये Quantity Order Entity का हिस्सा (Attribute) तो हो नहीं सकता क्योंकि Quantity उस Item पर Depend करती है, जिसे Music Store Order कर रहा है। इसी तरह से ये Quantity Item

Entity का भी हिस्सा (Attribute) नहीं हो सकता क्योंकि Quantity किसी Specific Order पर Depend करता है।

इस प्रकार के Attribute में Store होने वाले Data को ***Relationship Data*** कहा जाता है, जहां कोई Data किन्हीं Entities के Relationship का हिस्सा होता है ना कि वह Data Relationship में भाग लेने वाले किसी Entity का हिस्सा (Attribute) होता है।

चूंकि किसी Relationship में Attributes नहीं होते हैं। इसलिए हमें Relationship Data को Represent करने के लिए किसी ऐसे Entity की जरूरत होती है, जो दो Entities के बीच की Relationship को Represent करे और इस Entity में हम उस Relationship Data को एक Attribute के रूप में Store कर सकते हैं।

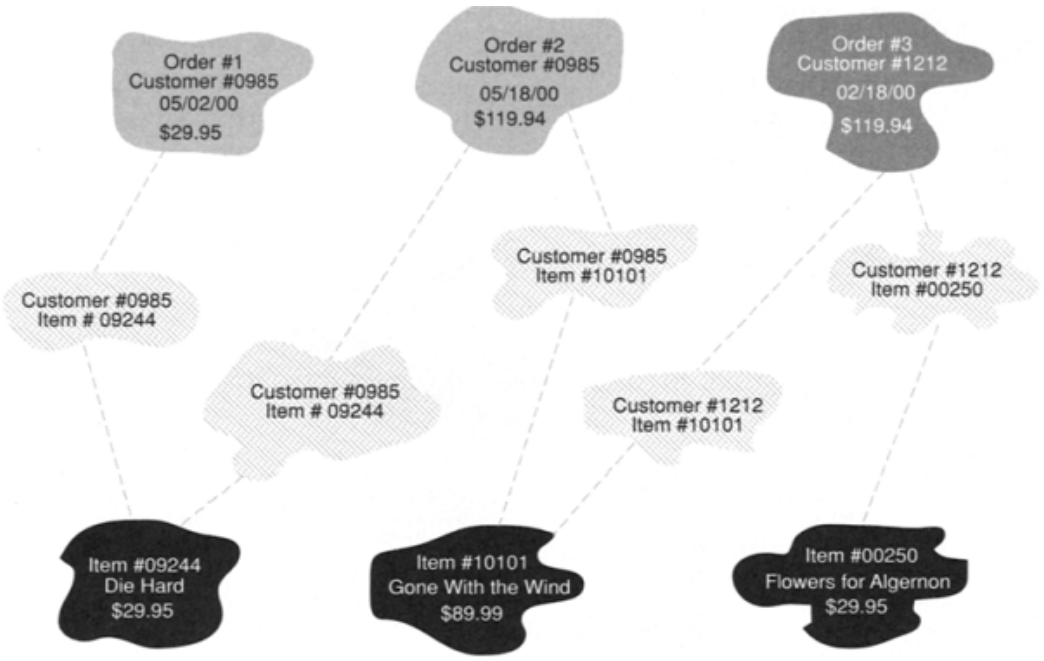
यानी जब किन्ही दो Entities के बीच कोई ऐसा Data हो जो किसी भी Entity में स्वतंत्र रूप से Represent नहीं हो सकता, लेकिन दोनों Related Entities के Group पर आधारित होता है, तब हमें इस प्रकार के **Relationship Data** को Store करने के लिए एक नए Entity की जरूरत पडती है और इस प्रकार के Data को उस नए Entity में Store किया जाता है।

Composite Entities

वे Entities, जो दो अन्य Entities के बीच की Relationship को Represent करने के लिए Describe किए जाते हैं, **Composite Entities** कहलाते हैं। Composite Entities किस प्रकार से काम करते हैं, इस बात को हम एक उदाहरण द्वारा समझने की कोशिश करते हैं।

हम समझ सकते हैं कि Music Store के Customers के Orders व Music Store के Items के बीच ***Many To Many*** की Relationship है क्योंकि एक Order में कई Items Appear हो सकते हैं और समान Item को एक से ज्यादा Order में Appear किया जा सकता है।

अब हमें एक ऐसे Entity की जरूरत होती है, जो हमें ये बताए कि एक Specific Title किसी Specific Order पर Appear हो रहा है। इस Concept को निम्न चित्र द्वारा समझाया जा रहा है:



इस चित्र में तीन Order Instances हैं और तीन ही Item Instances हैं। पहला Order (Order #1) Customer Number #0985 ने दिया है और इस Order में उसने केवल एक ही Item (item #09244) का Order दिया है।

दूसरे Order (Order #2) Customer #0985 ने दिया है और इस Order में उसने जिन दो Items का उल्लेख किया है, उनका Number #02944 व #10101 है। यानी दूसरे Order में Item Number #02944 तो फिर से Appear हो ही रहा है साथ ही Item Number #10101 भी Appear हो रहा है।

तीसरा Order जो कि Order #3 है, वह Customer Number #1212 ने Place किया है और इसमें भी दो Items Item Number #10101 व Item Number #00250 का Order दिया गया है।

इस चित्र में हम देख सकते हैं कि तीन Customers ने तीन Orders Place किए हैं और तीनों Orders में कुल पांच Items को Order किया गया है। इसका मतलब ये है कि यदि तीनों Orders को पूरे किए जाएं, तो Item Number #02944 की दो Copies, Item Number #10101 की दो Copies व Item Number #00250 की केवल एक Copy यानी कुल पांच Copies Sell होंगी।

इस Diagram के बीच का हिस्सा ही वह **Composite Entity** है, जो कि हर Order को किसी एक Specific Item से Connect कर रहा है। हमारे इस Example Diagram में कुल पांच Instances हैं, जिन्हें हम "**Line Items**" कहेंगे। ये **Line Item** Entities, **Order** व **Item** के बीच की Relationship को Represent करने के लिए Create किए गए हैं।

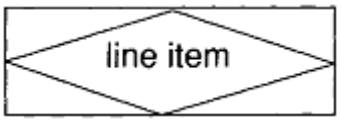
इस Diagram में हम देख सकते हैं कि हर Order प्रत्येक Item के लिए केवल एक **Line Item** Instance से Related है। दूसरे तरीके से देखें तो प्रत्येक Item हर Order पर Appear होने के लिए किसी एक **Line Item** Instance से Related है।

इस Diagram के आधार पर प्रत्येक Line Item Instance एक और सिर्फ एक ***Order*** से Related है साथ ही वही Line Item एक और सिर्फ एक ***Item*** से Related है। परिणामस्वरूप एक Order व उसके Line Items के बीच ***One To Many*** की Relationship है, क्योंकि एक Order कई Line Items से Related है और **Item** वह **Order,** जिस पर वह Item Appeared है, के बीच ***One To Many*** की Relationship है क्योंकि एक **Item** एक से ज्यादा **Line Items** पर Appear है।

इस तरह से इस Diagram में हम देख सकते हैं कि एक Composite Entity की उपस्थिति से Original ***Many To Many*** की Relationship दो ***One To Many*** की Relationship में Convert हो रही है। यदि हमारे Database से सम्बंधित कोई **Relationship Data** है, तो उस Data को Store करने के लिए हम इस Composite Entity में ही उस Data के लिए Appropriate Attribute Create कर सकते हैं।

उदाहरण के लिए Order किए गए Item की Quantity को Store करने के लिए इस Composite Entity में Quantity नाम का Field या Attribute Create कर सकते हैं। इसके साथ ही हम इस Composite Entity में इस बात की जानकारी रखने के लिए एक Flag Attribute Create कर सकते हैं, जो इस बात की जानकारी दे कि Order किए गए Items को Ship किया जा चुका है या नहीं और यदि Ship किया जा चुका है, तो Shipping Date को Store करने के लिए **Shipping Date** Attribute को भी इसी Composite Entity में एक Field के रूप में Specify किया जा सकता है।

Chen के ER Method में Composite Entity को ER Diagram में Draw करने के लिए निम्नानुसार Symbol का प्रयोग किया जाता है, जबकि Information Engineering Method में Composite Entity को ER Diagram में Represent करने का कोई तरीका नहीं है।



Music Store Database के सभी Many To Many Relationships को Eliminate करने के लिए हमें हर Many To Many Relationship को एक Composite Entity द्वारा दो One To Many Relationships में Convert करना होगा।

जैसाकि हम हमारे Music Store Database के पिछले IE ER Diagram में देख सकते हैं, उसमें तीन Many To Many Relationships हैं, इसलिए इन तीनों Many To Many Relationships को Remove करने के लिए हमें तीन Composite Entities को निम्नानुसार Create करना होगाः

1 **Order Lines**

Order Lines Entity एक Item को एक Order पर Represent करता है। हर Order की कई "**Order Lines**" हो सकती हैं जो कि अलग–अलग Items से Connect हों, लेकिन Order Line एक और सिर्फ एक Order पर Appear हो सकता है। इसी तरह से एक Order Line में एक और सिर्फ एक Item Appear हो सकता है लेकिन एक ही Item एक से ज्यादा Order Lines में Appear हो सकता है और प्रत्येक Item किसी Different Order से Connected होता है क्योंकि एक ही Item के लिए एक से ज्यादा Orders Place किए जा सकते हैं।

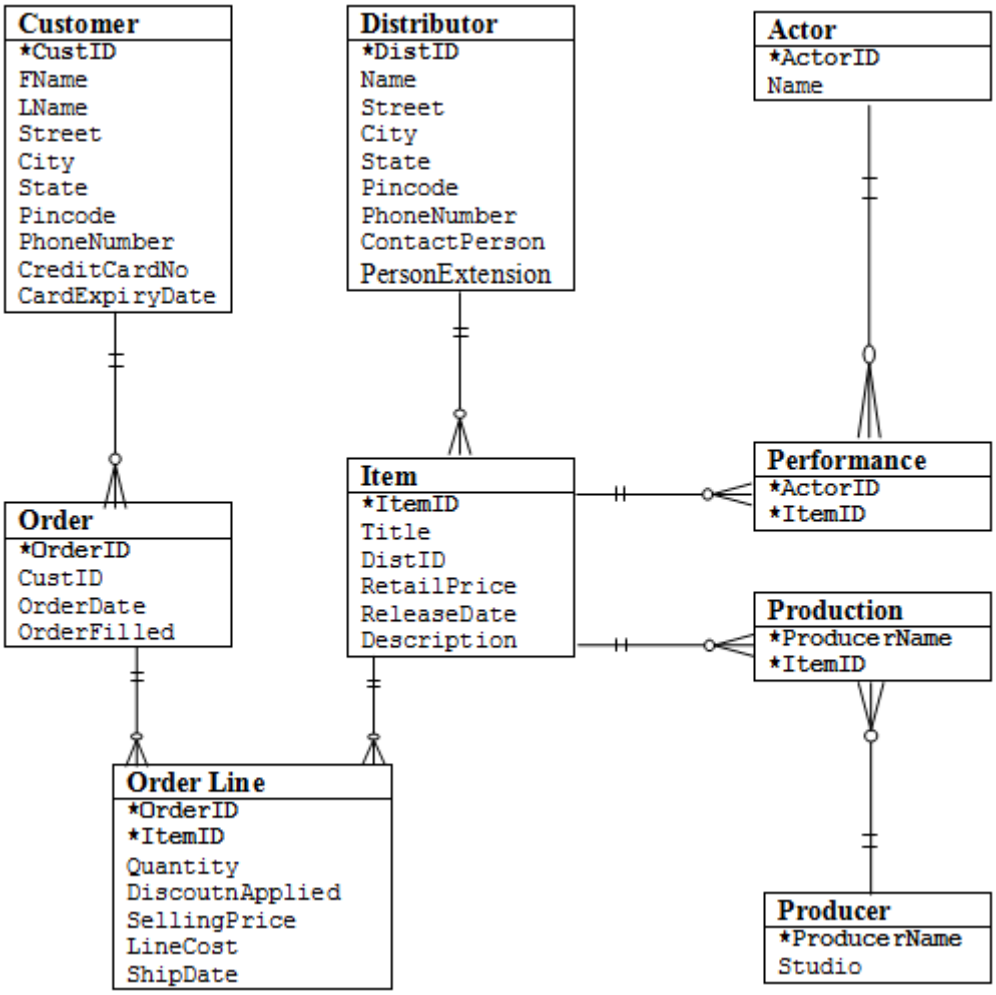
2 **Performance**

Performance Entity ये Represent करता है कि एक Actor एक Film में Appear होता है। हर Performance एक और सिर्फ एक Film के लिए होती है, जबकि एक Film में एक से ज्यादा Performance हो सकती हैं, क्योंकि एक Film में एक से ज्यादा Actor हो सकते हैं। इसी तरह से एक Actor प्रत्येक Film में एक Performance से Related होता है, लेकिन एक Film में केवल एक और सिर्फ एक Performance से Related होता है।

3 **Production**

Production Entity ये Represent करता है कि एक Producer एक Film पर काम करता है। एक Producer कई Productions में Involved हो सकता है जबकि हर Production केवल एक और सिर्फ एक Producer से Related होता है। Item के साथ की Relationship ये Indicate करता है कि हर Film को एक से ज्यादा Producers Produce कर सकते हैं लेकिन प्रत्येक Production किसी एक Item से Related होता है।

चूंकि **Composite Entities** को मुख्यतः दो Entities के बीच की Relationships को Indicate करने के लिए Create किया जाता है, इसलिए Composite Entity का उसके दोनों Child Entities से Related होना जरूरी होता है। ऐसा इसलिए होता है क्योंकि हर Child Entity का उसके Parent Entity से Related होना Compulsory या Mandatory होता है। अब बनाया जाने वाला Modified **IE Design** निम्नानुसार होगाः



Relationships and Business Rules

Database Design कई तरीकों से Science के साथ–साथ एक Art भी है। किसी Business के लिए किसी Database का कौनसा Design पूरी तरह से Correct होगा, ये उस Organization के Business Rules पर निर्भर होता है। अलग–अलग Organizations अपने अलग–अलग Business Rules के आधार पर काम करते हैं, इसलिए एक Organization के लिए Design किया गया Database कभी भी पूरी तरह से किसी दूसरे Organization के Database पर पूरी तरह से Apply नहीं किया जा सकता है।

उदाहरण के लिए मानलो कि हम किसी Retail Establishment के लिए एक Database Create करना चाहते हैं, जिसके बहुत सारे अन्य Stores हैं। इस Database में जो मुख्य बात Handle की जाएगी, वह उन Employees का Schedule होगा, जो उस Retail Establishment के विभिन्न Stores पर काम करते हैं।

इससे पहले कि हम ऐसा Database Design करें, हमें Employee व Store के बीच की Relationship को Identify करना होगा। Employee व Store के बीच कौनसी Relationship होगी? One To One की या One To Many की।

क्योंकि Retail Establishment का Chairman कभी भी किसी भी Employee को अपने किसी भी Store पर भेज सकता है। ऐसा जरूरी नहीं है कि जो Employee जिस Store के लिए नियुक्त किया जाएगा, वह हमेंशा उसी Store पर नियुक्त रहेगा। हर Employee हमेंशा एक ही Store के लिए नियुक्त किया जाता है, तो Employee व Store के बीच One To Many की Relationship Identify होती है और यदि हर Employee प्रत्येक Store पर थोडा–थोडा समय व्यतीत करता है, तो Employee व Store के बीच Many To Many की Relationship Identify होता है।

इस स्थिति में मामला ये नहीं होता है कि कौनसा Design सही है और कौनसा गलत बल्कि इस स्थिति में कौनसा Design सही है, ये बात उस Organization के काम करने के तरीके पर निर्भर करता है, कि वह Organization अपने Business को किस तरह से Operate करता है।

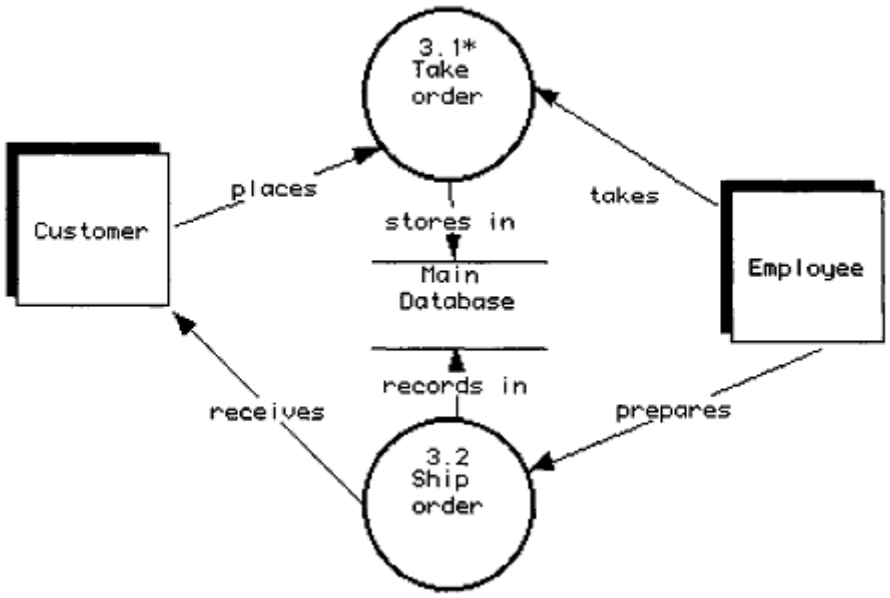
यानी इस बात से कोई फर्क नहीं पडता है कि हम Database Design के सम्बंध में कितना ज्यादा जानते हैं, बल्कि हम तब तक एक अच्छा Database Create नहीं कर सकते हैं, जब तक हमारा Database किसी Business Environment की विभिन्न Relationships को Accurate तरीके से Reflect नहीं करता है।

Data Modeling and Data Flow

Data Model Design करते समय जो सबसे ज्यादा Common Mistakes की जाती हैं, वो Data Model व Data Flows के बीच Confusion की Mistake होती हैं। **Data Flow** इस बात को Show करता है कि किसी Organization में Data को किस प्रकार से Handle किया जाता है, Data को कहां Store किया जाता है और Data के साथ क्या Processing की जाती है।

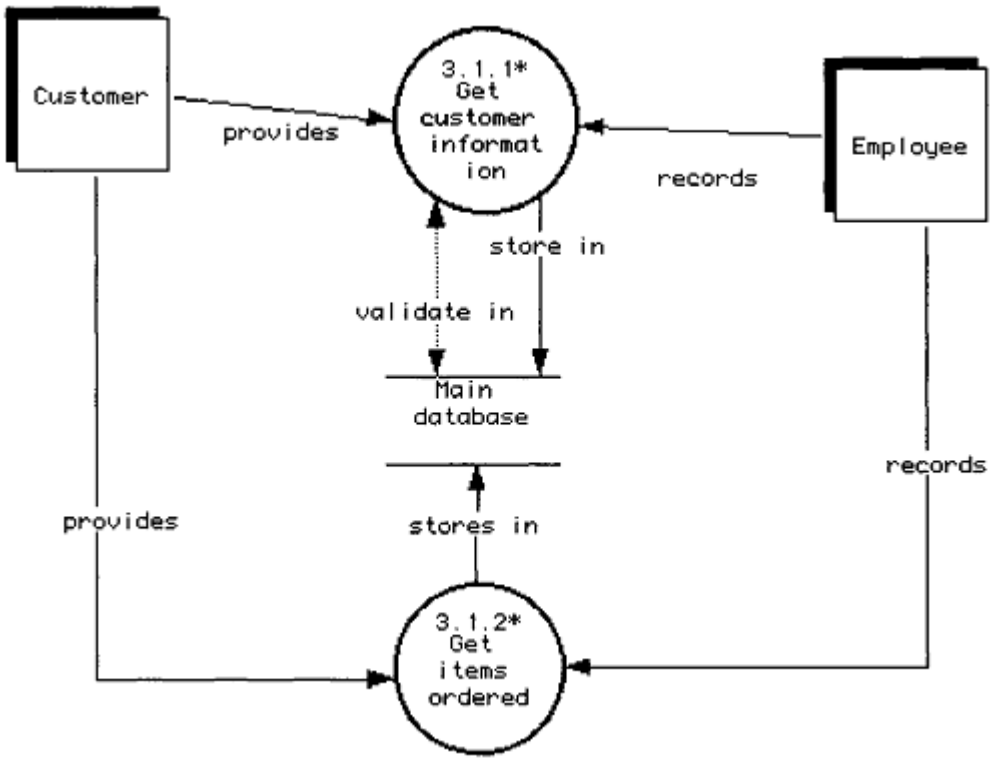
जबकि **Data Medal** Data की Internal बातों को तथा Data के बीच की आपसी Relationships को बिना इस बात की परवाह किए Represent करने का काम करता है, कि Data को कौन Handle कर रहा है और Data के साथ किस प्रकार की Processing को Apply किया जा रहा है।

Data Flows को सामान्यतया Data Flow Diagrams(DFD) में Document किया जाता है। उदाहरण के लिए अगले चित्र में हम Music Store Organization के Top Level Data Flow Diagram को देख सकते हैं।



इस चित्र का Sequence उन लोगों को प्रदर्शित कर रहा है कि जो Data को Handle कर रहे हैं। इस चित्र के Circles उन Processes को Represent कर रहे हैं, जिन्हें Data पर Apply किया जाता है। जिस जगह पर Data को Store किया जाता है, उसे दो Parallel Lines के रूप में दर्शाया गया है। हमारे इस उदाहरण में मुख्य Storage को "**Main Database**" शब्द से Represent किया गया है। इस चित्र में दर्शाए गए Arrows इस बात को Indicate करते हैं कि Data किसी स्थान से किस स्थान की तरह Flow या Pass किए जा रहे हैं।

Data Flow Diagrams का प्रयोग सामान्यतया Design किए जा रहे System की और अधिक Details Provide करने के लिए किया जाता है। पिछले चित्र के Order लेने की प्रक्रिया को अगले चित्र में थोडा सा और Modify करके "**Take Order**" की Process को Represent किया गया है।



इस चित्र में हम देख सकते हैं कि एक Order लेने की प्रक्रिया में दो बडी बातें Involved हैं। पहली Customer की Information लेना व दूसरी Order किए जाने वाले Items की Information लेना। इस चित्र के हर Process को और Explore करके हम Data Flow की और अधिक Deep Details प्राप्त कर सकते हैं, जैसाकि अगले दो चित्रों में दर्शाया गया है।

इस स्थिति में Diagrams किसी System की इतनी Detailed Information दे देते हैं, कि एक Application Designer उस Application को Design करने का Plan बना सकता है। Data Flow व Data Model दोनों को Separate रखने के लिए हम कुछ Guide Lines का प्रयोग कर सकते हैं। ये Guide Lines निम्नानुसार हैं:

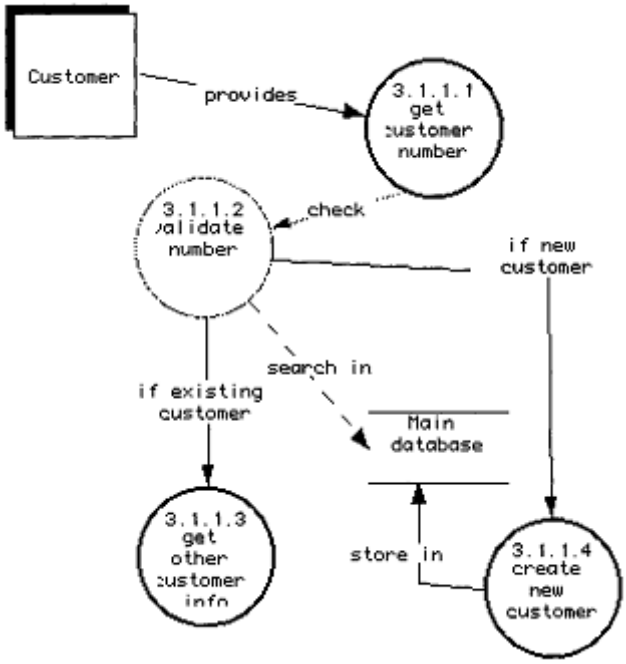
1 एक Data Flow Diagram ये Show करता है कि Data को कौन Use या Handle कर रहा है, जबकि Data Model ऐसा नहीं करता है।

2 एक Data Flow Diagram ये Show करता है कि Business Related Data को किस तरह से व किन माध्यमों (People Inquiry etc.) से Collect किया गया है, जबकि Data Model इस बात की जानकारी नहीं देता है।

3 एक Data Flow Diagram Data पर Perform होने वाले उन Operations को Show करता है, जो Data को एक रूप से दूसरे रूप में Transform करते हैं, जबकि Data Model इस बात को Show नहीं करता है।

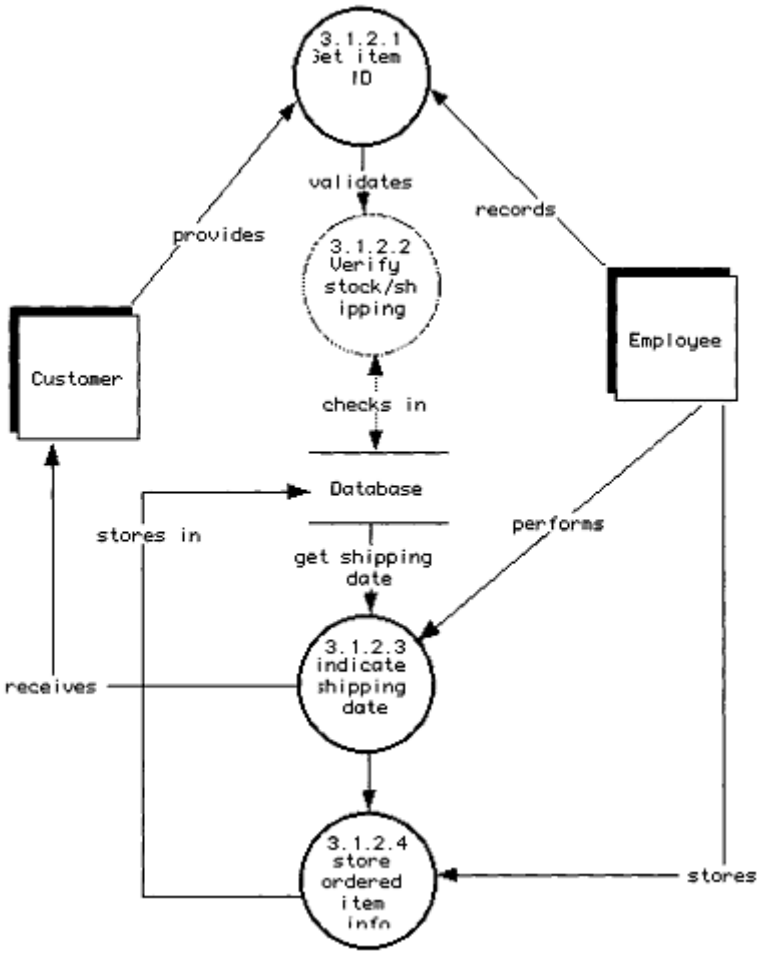
4 एक Data Model ये Show करता है कि किसी Database के विभिन्न Entities किस प्रकार से आपस में Internally Related हैं, जबकि Data Flow Diagram इस बात की कोई जानकारी नहीं देता है।

5 एक Data Model उन Attributes को Show करता है, जो किसी Data Entity को Describe करते हैं, जबकि Data Flow Diagram किसी Entity के Attributes की कोई जानकारी नहीं देता है।



एक Data Model में किसी Database में Store होने वाले Data जैसे कि Entities, Attributes व Entity Relationships की जानकारी होती है। यदि किसी Entity के Data को किसी Database में Store नहीं किया जा रहा हो, तो वह Entity उस Database का हिस्सा नहीं होता है।

उदाहरण के लिए हालांकि Music Store का Data Flow Diagram Music Store के उस Employee को Show करता है, जो Music Store के विभिन्न प्रकार के Data को Handle करता है, लेकिन उस Employee से सम्बंधित किसी प्रकार के Data को Database में Store नहीं किया जा रहा है। इसलिए Music Store Database के ER Diagram में Employee नाम का कोई Entity नहीं है।



DBMS

THE SCHEMA

DMBS – The Schema

एक Completed Entity-Relationship Diagram किसी Database के Overall Logical Plan को Represent करता है। Database Management System की भाषा में इस Overall Plan को **Schema** कहा जाता है। यही वह तरीका या Design होता है, जिसमें किसी Database को Maintain करने वाले लोग किसी Business System को देखते हैं।

वे Users जो इस Schema पर आधारित Database Application को Use करते हैं और वे Users जो इस Database Schema के आधार पर Application Develop करते हैं, वे दोनों ही इस Design के केवल Logical Schema तक से ही परिचित होते हैं। **Data** Physically किस प्रकार से Store होते हैं, इस बात की जानकारी इन दोनों को ही नहीं होती है, ना तो इस Logical Schema को ना ही इस Logical Schema को Use करने वाले Users को।

Logical Schema की Layer के नीचे Data का Physical Storage होता है, जिसे **DBMS** Manage करता है। इसे **Physical Schema** कहा जाता है। Physical Schema को DBMS ही Handle करता है। केवल कोई बहुत ही बडा DBMS हमें ये सुविधा देता है कि हम Physical Schema को Control कर सकें।

इस तरीके का सबसे बडा फायदा ये है कि Database Design करने वाले व उसे Use करने वाले, दोनों को ही ये जानने की जरूरत नहीं होती है कि Data Physically किस प्रकार से Store हो रहा है। इस तरीके के कारण Database को Access करना काफी सरल हो जाता है क्योंकि हम बहुत ही आसानी से Logical व Physical Schemas को Change कर सकते हैं।

चूंकि हम एक Database को तीन तरीकों से देख सकते हैं, पहला Logical Schema के रूप में, दूसरा User के रूप में व तीसरा Physical Schema के रूप में, इसलिए आजकल कुछ Database को Three-Schema Architecture भी कहा जाने लगा है। System Programmers व अन्य लोग जो कि Physical Storage को Manage करते हैं, वे वास्तव में Physical Schema के साथ Deal करते हैं। आज हमारे सामने जितने भी DBMS Softwares हैं, वे हमें उस DBMS में Store होने वाले Data के File Structure को Control करने की कोई सुविधा प्रदान नहीं करते हैं।

Database Designers, Database Administrators व कुछ Application Programmers Logical Schema को Use करते हैं। End User Interactively काम करते हैं, यानी पहले से बने हुए Database System को Application के माध्यम से Use करते हैं जबकि Application Programmers Database को User View के आधार पर देखते हैं और End User के लिए Database Applications Create करते हैं।

जब हम एक बार ER Diagram Complete कर लेते हैं, उसके बाद Create होने वाले Conceptual Logical Schema को Use किए जाने वाले DBMS के आधार पर Formal Data

Model में Translate करना होता है। आज जितने भी DBMS Softwares Available हैं, वे सभी Relational Data Model पर आधारित हैं। Relational Database एक ऐसा Database होता है, जिसका Logical Structure Relations के एक Collections के अलावा कुछ नहीं होता है।

Relational Database Model को Edgar (E. F.) Codd ने Develop किया था। 1960 के दंशक में Dr. Codd Existing Data Models पर काम कर रहे थे। अपने अनुभव के आधार पर उन्होंने पाया कि जितने भी Data Models उस समय प्रचलित थे, वे सभी काफी जटिल व अप्राकृतिक तरीकों से Data को Model करते थे। चूंकि वे एक गणितज्ञ थे, इसलिए उन्होंने विभिन्न प्रकार के Relations को Set Theory के आधार पर Mathematical Form में Develop करना शुरू किया और अपने Concept को और Extend करके उन्होंने **Relational Database Model** Develop किया और 1970 में लोगों के सामने लाया।

Mathematical Set Theory में Rows (***Tuple***) व Columns (***Attributes***) से बनी एक **Table** को एक **Relation** के रूप में Define किया जाता है। Relation को दूसरे शब्दों में हम Table भी कह सकते हैं। ये Definition केवल इस बात को Specify करता है कि किसी Table के हर Column में क्या Store किया जाएगा, लेकिन उसमें Actual Data को Specify नहीं किया जाता है। जब हम इस Table में Data के Rows Include करते हैं, तब हमें उस Relation का एक Instance प्राप्त होता है। उदाहरण के लिए हम किसी Student के Relation को निम्नानुसार Represent कर सकते हैं:

SrNo	Name	FName	City	Dist.	Class	DOB	DOA
001	Rahul	Mohan Lal	Falna	Pali	10	10-02-1982	15-7-1987
002	Rohit	Sohan Lal	Bali	Pali	09	11-12-1983	05-7-1987
003	Krishna	Gopal	Desuri	Pali	08	20-03-1981	10-7-1987
004	Madhav	Ram Lal	Falna	Pali	10	30-2-1982	01-7-1987
005	Achyut	Nand Lal	Desuri	Pali	07	12-12-1986	13-7-1987
006	Manohar	Rohan Lal	Bali	Pali	10	10-11-1982	15-7-1987

पहली नजर में ये Relation एक Flat File या किसी Spreadsheet के Rectangular Portion की तरह ही दिखाई देता है। लेकिन जब हम इस Table को Set Theory के आधार पर एक Relation के रूप में देखते हैं, तब इस Relation के कुछ बहुत ही Specific Characteristics हमें दिखाई देते हैं। Set Theory के आधार पर देखने पर इस Relation का हर Column DBMS में Store किए जाने वाले Constraints को Represent करता है।

जब हम Set Theory के आधार पर किसी Relation के एक Column की Characteristics को देखते हैं, तब हमें एक Column की निम्न Characteristics होती हैं:

1 **एक Table में हर Column का एक Unique नाम होता है।**
किसी एक ही Relation Schema में दो या दो से अधिक Tables में एक ही नाम के Columns हो सकते हैं, लेकिन किसी एक ही Table में एक ही नाम के दो Column नहीं हो सकते हैं।

जब समान नाम का कोई Column एक से ज्यादा Tables में Appear होता है और समान नाम के Columns को Hold करने वाले Tables जब समान Data Manipulation Operation के लिए Use किए जाते हैं, तब हमें उन समान नाम के Columns को Particular Table से Access करने के लिए उनके Tables के नाम को उपसर्ग के रूप में Columns के नाम से पहले Use करना पडता है और Columns के नाम को Table के नाम से एक Dot या Period द्वारा जोडना पडता है। जैसेः

```
Customers.CustID
Students.StudID
```

2 **एक Column हमेंशा सिर्फ एक ही Domain के मान को Store करता है।**
किसी Table में जिस Column को जिस Type का मान Store करने के लिए Define किया जाता है, वह Column केवल उसी मान को Store करता है।

उदाहरण के लिए किसी Student Table में Student का नाम Store करने के लिए जिस Column को Define किया गया है, वह Column केवल नाम Store करने के लिए ही Use किया जा सकता है, जबकि जो Column Roll Number Store करने के लिए Define किया गया है, उसमें केवल Roll Number ही Store किया जा सकता है। यानी हर Column में किसी एक निश्चित Domain के मान को ही Store किया जा सकता है।

परिणामस्वरूप Relations को **Column Homogeneous** कहा जाता है। साथ ही Table का हर Column किसी ना किसी Domain Constraint से Associated होता है। हमारे DBMS के आधार पर Domain Constraint Data Type की तरह ही Simple होता है, जैसे कि Integers, Characters, Date आदि।

इसके अलावा हमारा DBMS इस बात की भी सुविधा देता है कि हम हमारी जरूरत के आधार पर नया Domain भी Create कर सकते हैं और उसे अपनी Table के किसी Column के साथ Attach कर सकते हैं।

Columns की तरह ही Rows की भी किसी Relation में अपनी कुछ Special Properties या Characteristics होती हैं। ये Characteristics निम्नानुसार हैंः

1 एक Row के विभिन्न Column में हम सिर्फ और सिर्फ एक ही मान को Store कर सकते हैं। यानी किसी Row का हर Column **Single-Valued** होता है। और

2 एक Relation में हर एक Record **Unique** होता है। यानी एक Relation में एक Row का दुबारां Duplication नहीं हो सकता। किसी Relation के हर Record को Unique

बनाने के लिए DBMS स्वयं ही हर ROW के साथ Unique Constraint को Enforce नहीं करता है, बल्कि इस सुविधा को हम **Primary Key** द्वारा प्राप्त करते हैं।

3 **Primary Key** किसी Table के किसी Column या Columns का Combination होता है, जिसे उस Table या Relation के किसी Record या Row को Uniquely Identify करने के लिए Define किया जाता है। जैसे ही किसी Relation में कोई **Unique Primary Key** को किसी विशिष्ट Column या Columns के Combination के साथ Set कर दिया जाता है, वैसे ही ये निश्चित हो जाता है, कि उस Table का हर Record या Row Unique होगा।

Tables

एक Relational Database दो तरह के Tables के साथ प्रक्रिया करता है, जिन्हें **Base Table** व **Relation** कहते हैं। ये दोनों ही Database में Store होते हैं। ये ही वे Tables होते हैं, जो हमारे Database का **Conceptual Logical Schema** बनाते हैं।

इसके अलावा Tables पर जिन Relational Operations को Perform किया जाता है, उनके परिणामस्वरूप कुछ Additional Tables Produce होते हैं। ये Tables सिर्फ **RAM** या Main Memory में Exist होकर अपना काम पूरा करते हैं और फिर Destroy हो जाते हैं, इसलिए इन्हें **Virtual Tables** कहा जाता है। Virtual Tables एक Legal Relation नहीं होते हैं, क्योंकि इनमें Primary Keys को Define नहीं किया जा सकता है। लेकिन चूंकि Virtual Tables Database में Store नहीं होते हैं, इसलिए ये Database Design में किसी तरह की कोई समस्या पैदा नहीं करते हैं।

Virtual Tables DBMS को कई तरीकों से फायदा पहुंचाते हैं। सबसे पहले तो ये Tables DBMS को Application की Processing के दौरान Generate होने वाले Intermediate Query Result को Database में Store करने के बजाय इन Virtual Tables में Store करके Main Memory में Store करने की सुविधा देते हैं, जिससे Query की Performance अच्छी हो जाती है, क्योंकि Main Memory की Speed हमेंशा ही Disk की Speed से ज्यादा होती है।

DBMS का दूसरा फायदा ये होता है कि जो Tables Relational Data Model के Rules को Violate करते हैं, DBMS उन्हें Virtual Tables के रूप में Main Memory में Store करके Manage करता है, जिससे Actual Database में Stored Data की सुरक्षा को किसी प्रकार का कोई खतरा नहीं रहता।

तीसरा और अन्तिम फायदा ये होता है कि DBMS द्वारा Virtual Tables के प्रयोग के कारण बार–बार Disk पर Write/Read/Delete Operations को Perform नहीं करना पडता है, जिससे Disk पर Data Fragments या कई टुकडों में Store नहीं होता है। इससे समय की भी बचत होती

है और Database का Performance भी अच्छा हो जाता है। Virtual Tables को सामान्यतया Temporary Tables या Temporary Base Tables भी कहा जाता है।

किसी Relation को Represent करते समय उसमें Data को Store नहीं किया जाता है। इस स्थिति में किसी Relation को Represent करने का एक Common तरीका निम्नानुसार होता है:

```
RelationName ( PrimaryKeyColumn, NonPrimaryKeyColumn1, ... , NonPrimaryKeyColumnN)
```

उदाहरण के लिए यदि हम किसी Customer Relation को Represent करना चाहें, तो निम्नानुसार कर सकते हैं:

```
Customers (CustID, FName, LName, Phone)
```

इसी तरह से यदि हम किसी Student के Relation को Represent करना चाहें, तो उसे भी निम्नानुसार Represent कर सकते हैं:

```
Students (SrNo, Name, FName, Add, City, State, DOB, DOJ)
```

ये दोनों Expressions किसी Relation के Structure को Represent करने के Ideal Expressions हैं, जिनमें कोई Data नहीं है। यदि किसी Relation में Data Included हों, तो ऐसा Relation वह Expression उस Relation का एक Instance होता है।

Primary Keys

जैसाकि हमने पहले भी बताया कि **Primary Key** किसी Table कि प्रत्येक Record या Row को Uniquely Identify करना सम्भव बनाता है। किसी Relation में Primary Key की वही भूमिका होती है, जो किसी Entity Identifier की होती है। Primary Key को Define करके हम ये तय करते हैं कि हमने जिस छोटे से छोटे Data को Database में Store किया है, हम उस छोटे से छोटे Data के टुकडे को भी फिर से प्राप्त कर सकेंगे।

जब हम Relational Database की बात करते हैं, तब किसी Database से किसी Data के छोटे से छोटे टुकडे को भी प्राप्त करने के लिए हमें सिर्फ तीन बातों की जानकारी होनी चाहिए: उस Table के नाम की, जिसमें Data Stored है, उस Column के नाम की, जिससे हम Data को प्राप्त करना चाहते हैं और उस Row के Primary Key की, जिसके Column के Data को हम प्राप्त करना चाहते हैं।

यदि हर Row के लिए Primary Key Unique हो, तो हम इस बात के लिए Sure हो सकते हैं कि हम Exact Row से Data को Retrieve कर सकेंगे। लेकिन यदि हर Row Unique ना हो, तो

हम एक से ज्यादा Rows को Retrieve करेंगे, जहां ये तय करना मुश्किल हो जाएगा, कि हमारा Required Data किस Row में है।

चूंकि एक Primary Key हमेंशा किसी Record या Row को Uniquely Identify करने के लिए Define किया जाता है, इसलिए किसी भी Primary Key Column को Empty नहीं छोडा जा सकता या किसी Primary Key Field में NULL Store नहीं किया जा सकता है। Null एक Special Database मान होता है, जिसका मतलब "Unknown" होता है।

यदि हमारे Relation में केवल एक ही Record हो और हम Primary Key Filed को Null कर दें, तो कोई Problem Generate नहीं होती है, लेकिन जैसे ही हम उस Relation में दूसरा Record Enter करेंगे, हम Primary Key Fields के Uniqueness की Property को खो देंगे। इसलिए हम कभी भी किसी Table के Primary Key Field को Null Assign नहीं कर सकते हैं।

इस Constraint को **Entity Integrity** कहा जाता है और Database में जितनी बार भी Data Enter या Modify किया जाता है, DBMS Primary Key को Enforce करके Database पर Apply करता है, जिससे हर Record हमेंशा Unique बना रहता है।

हम किसी Primary Key Field में Duplicate Data Store नहीं कर सकते हैं। यदि हम ऐसा करने की कोशिश करते हैं, तो भी DBMS हमें ऐसा नहीं करने देता है, क्योंकि वह Primary Key के Constraint को Follow करता है, जिसके तहत एक Column में कभी भी Duplicate Values Store नहीं हो सकती है।

किसी Relation के लिए एक उचित Primary Key Select करना एक Challenging काम होता है। कुछ Entities में Natural Primary Keys होती हैं, जैसे कि Purchase Order या Invoice को Identify करने के लिए हमेंशा एक Meaningless Unique Number होता है, जिससे कोई Particular Purchase Order या Invoice Identify होता है। इसे ही **Natural Primary Key** कहा जाता है और ये ही एक Ideal Primary Key का उदाहरण होता है।

हम किसी भी Entity के उसी Attribute को किसी Relation में Primary Key के रूप में Identify कर सकते हैं, जिसकी निम्नानुसार दो विशेषताएं हों:

- Primary Key की Value कभी भी Change नहीं होती है। **(Uniqueness)**
- Primary Key Column को **Null** नहीं रखा जा सकता है। **(Not Null)**

हम किसी Entity के किसी ऐसे Field को Primary Key Set नहीं कर सकते हैं, जिनका Repetition सम्भव हो। उदाहरण के लिए यदि हम किसी Customer Relations में Customer के Name Filed को Primary Key Set कर दें, तो DBMS में Enter किए एक नाम के एक ही

व्यक्ति को Database में Store होने देगा। क्योंकि DBMS Primary Key Field में एक ही Value का Repetition नहीं करने देता है, जबकि एक ही नाम के दो Customer हो सकते हैं।

इसी तरह से हम उस Attribute को भी Primary Key Set नहीं कर सकते हैं, जो बार–बार Change होता है। इस स्थिति में किसी एक ही Entity के एक ही Instance को एक ही Relation में एक से ज्यादा बार Enter किया जा सकता है, जिससे Database की Consistency व Accuracy प्रभावित होती है। इस सम्बंध में हम पहले भी चर्चा कर चुके हैं।

एक Appropriate Primary Key को Choose करने के लिए हमें हमेंशा किसी Entity के Meaningful Attributes को Avoid करना चाहिए। किसी Meaningful Information को Code के रूप में Specify करके उसे किसी Relation में Primary Key के रूप में Use नहीं करना चाहिए, जैसाकि हमने Music Store Database में किया था।

इस तरह के Code को Decode करने में समस्याएं पैदा हो सकती हैं या इस तरह के Codes को बनाने में भी गलतियों की सम्भावना रहती है। साथ ही Meaningful Information Change होने की भी सम्भावना रहती है, जैसाकि Music Store Database के सन्दर्भ में बताया गया है। किसी Primary Key के उपरोक्त दो गुणों के अलावा ये Primary Key का तीसरा गुण होता है किः

- एक Primary Key के रूप में Meaningful Data को Avoid करना चाहिए।

हालांकि जरूरत के आधार पर व अच्छी तरह से सोंच–विचार कर लेने के बाद यदि हमें लगता है, कि किसी Meaningful Data को Primary Key बनाया जा सकता है, तो हम किसी Meaningful Data को भी Primary Key के रूप में Specify कर सकते हैं।

कई बार स्थितियां भी ऐसी ही होती हैं, जहां पर किसी विशेष अर्थ वाले Meaningful Data को ही Primary Key बनाना जरूरी होता है। उदाहरण के लिए यदि हमें किसी घटना को Time या Date के साथ Specify करना हो, तो हमें Date या Time जैसे Meaningful Data को Primary Key के रूप में Use करना जरूरी हो जाएगा।

Composite Keys

कई Tables ऐसे होते हैं, जिनमें कोई भी ऐसा Single Column नहीं होता है, जिसमें Values का Duplication ना हो। उदाहरण के लिए यदि हम Order Lines के Table को देखें, तो चूंकि एक Order में एक से ज्यादा Items Appear हो सकते हैं, इसलिए Order Numbers भी एक से ज्यादा बार Repeat होता है, इसी तरह से एक ही Item एक से ज्यादा Order पर Appear हो सकता है, इसलिए Item Number भी एक से ज्यादा बार Repeat होता है, जैसाकि हम निम्न Table में देख सकते हैं। इसलिए इस Table का कोई भी एक Single Column Primary Key की तरह Use नहीं किया जा सकता।

OrderID	ItemID	Quantity
10999	1122	1
10999	2211	3
10999	1002	1
10990	1122	2
10990	2211	4
10993	1122	1
10993	1100	2
10995	1100	1

फिर भी यदि हम Order Number व Item Number को Combined Form में Use करें, तो ये किसी Row या Record को Uniquely Identify कर सकते हैं। जब किसी Table में दो या दो से ज्यादा Columns को Combined रूप से Primary Key के रूप में Use करके किसी Record या Row को Uniquely Identify करते हैं, तो इस Columns के Group को **Composite Key** कहा जाता है।

हालांकि यदि हम चाहें तो इस Table के तीनों ही Columns को Combined Form में Use करके किसी Record या Row को Uniquely Identify कर सकते हैं, लेकिन इस Table में किसी Record को Uniquely Identify करने के लिए केवल Order Number व Item Number का Combination ही पर्याप्त है, इसलिए इस Composite Key में Quantity Column को Add करने की जरूरत नहीं है।

जब हम Composite Key का प्रयोग करके किसी Record को Uniquely Identify करना चाहते हैं, तब हमें सरलता के लिए कुछ बातों को ध्यान में रखना चाहिए, जिससे Database Design में किसी तरह की कोई परेशानी पैदा ना हो। ये बातें निम्नानुसार हैं:

1. एक Composite Primary Key में जहां तक सम्भव हो, कम से कम Columns का प्रयोग करना चाहिए। यानी यदि दो Columns के Group से Table के हर Record को Uniquely Identify किया जा सकता है, तो तीन Columns को मिलाकर Primary Key नहीं बनाना चाहिए।

2. जहां तक सम्भव हो, Composite Primary Keys को Meaningless रखना चाहिए।

हम Relations भी Create कर सकते हैं, जिसके सभी Columns को मिलाकर एक Key बना लिया गया हो। उदाहरण के लिए हम एक Library Card Catalog को लेते हैं। किसी Library में जितनी भी Books होती हैं, उन सभी Books का एक Unique ISBN (International Standard Book Number) होता है।

किसी Library Catalog में सभी ISBN Number की Book के साथ एक Subject Heading या विषय को Associate किया गया होता है और हर Book के साथ एक Subject Heading या विषय को Associate किया गया होता है। इस तरह से एक Book व उसके Subject Heading के बीच ***Many To Many*** की Relationship Create हो जाती है। इस Relationship को हम निम्नानुसार Represent कर सकते हैं:

SubjectCatalog (**ISBN**, **SubjectHeading**)

इस Relationship को Create करने के लिए हमें केवल हर **Subject Heading** को एक **Book Identifier** के साथ एक Pair के रूप में Specify करना होता है। इस स्थिति में इस Table के दोनों Columns Primary Key के हिस्से बन जाते हैं। हम देख सकते हैं कि इस Table के सभी Fields को Composite Primary Key के रूप में Use कर लिया गया है। इस तरह की Relationship से Database के Design में कोई Problem नहीं होती है। वास्तव में जब भी किसी Database में कोई Composite Entity होता है, जिसमें कोई **Relationship Data** नहीं होता, तब इस प्रकार की Relationships को Define करने पर Design सम्बंधित किसी प्रकार की कोई समस्या पैदा नहीं होती है। इस प्रकार के Composite Entity को सामान्यतया ***Many To Many*** की Relationship को Represent करने के लिए बनाया जाता है।

Representing Data Relationships

पिछले कई उदाहरणों में हमने विभिन्न प्रकार के Primary Key Identifiers को Use किया है। इन Keys के आधार पर ही Relational Database विभिन्न Entities के बीच की Relationships को Represent करता है। इस Concept को Clear करने के लिए निम्न तीन Tables को देखिए, जिनमें Data को Fill किया गया है:

Items Table

ItemID	Title	DistID	Price
2001	C in Hindi	200	200.0
2002	C++ in Hindi	200	225.0
2003	Java in Hindi	300	300.0
2004	DBMS in Hindi	400	150.0

Orders Table

OrderID	CustID	OrderDate
600000	00001	12/02/2006
600001	01000	15/06/2007
600002	00100	20/12/2008

Orders Lines Table

OrderID	ItemID	Quantity	Shipped?
600000	2001	1	Y
600000	2002	1	Y
600001	2002	2	Y
600002	2002	1	N
600002	2003	2	N
600002	2001	1	N

यहां Describe की गई सभी Tables "**Music Store"** Database के ER Diagram के समान ही हैं। यहां **Orders Table** (Orders Entity) के हर Unique Instance को **OrderID** नाम के एक Primary Key द्वारा Identify किया गया है, जो कि एक Meaningless Data है। **Items Table** (Items Entity) में हर Item को एक Unique Item Number द्वारा Identify किया जाता है, जिसे **ItemID** नाम दिया है और यहां भी ये एक Meaningless Data है। तीसरी Table Order Lines (Order Lines Entity) है, जो Music Store को ये बताता है कि कौनसा Item किस Order का हिस्सा है।

जैसाकि हमने Composite Entity के बारे में बताया, ये एक Composite Entity है और इस Table को एक Composite Primary Key की जरूरत होती है, क्योंकि Multiple Orders पर Multiple Items Appear हो सकते हैं।

इस Composite Primary Key का उन अन्य Primary Keys की तुलना में अधिक महत्व है, जिन्हें किसी Table के हर Instance या Row या Record को Uniquely Identify करने के लिए Define किया जाता है। इस Table में ये Composite Key हर Row को Uniquely Identify करने के साथ ही **Order Lines**, **Orders** व **Items** के बीच की Relationship को भी Represent करता है।

Order Lines Relation में **ItemID** Column उसी तरह का **Primary Key** है, जिस तरह का Item Table पर है। ये Primary Key दो Tables के बीच ***One To Many*** की Relationship को Represent करता है।

इसी तरह से **Orders** व **Order Lines** के बीच भी एक ***One To Many*** की Relationship है, क्योंकि **Order Lines** Table का **OrderID** उसी तरह का Primary Key है, जिस तरह का **Orders** Table में है।

जब किसी Table में वैसा ही Primary Key होता है, जैसा किसी दूसरी Table में होता है, तो इस प्रकार के Key को ***Foreign Key*** कहा जाता है। किसी Relational Database में किसी **Foreign Key** का किसी **Primary Key** के साथ Connect या Match होना, **Relationship** को Represent करता है। किसी Relational Database में Columns या Keys की Matching

के अलावा ऐसा कोई Structure नहीं होता है, जिससे Relationship को Represent किया जा सके।

यानी किसी Relational Database में Columns या Keys की Matching से ही विभिन्न Entities के बीच की आपसी Relationship को Represent किया जाता है। किसी Relational Database में विभिन्न Tables के बीच की Relationship Logical स्तर पर ही होती है, इस प्रकार की Relationship का Physical स्तर पर कोई अस्तित्व नहीं होता है।

Foreign Keys किसी Composite Primary Key का हिस्सा हो सकते हैं या वे उनके Table के Primary Key का हिस्सा नहीं भी हो सकते हैं। यानी मानलो कि Music Store के Customers व Orders के बीच निम्न Relation हैः

Customers (**CustID**, FName, LName, Telephone)
Orders (**OrderID**, *CustID*, OrderDate)

इस Relationship में **Orders** Table में जो **CustID** Column है, वह **Foreign Key** है, जिसे Customer Table के **Primary Key CustID** से Match किया गया है। यहां **Customers** व **Orders** के बीच ***One To Many*** की Relationship Represent हो रही है। फिर भी **Orders** Table का **CustID** Column **Orders** Table के **Primary Key** का हिस्सा नहीं है, बल्कि ये एक Non-Key Attribute है, इसलिए ये एक **Foreign Key** है।

तकनीकी रूप से Foreign Keys में तब तक किसी मान को Store नहीं किया जा सकता है, जब तक कि वे किसी Composite Primary Key का हिस्सा नहीं होते हैं। इन्हें Null Assign किया जा सकता है।

लेकिन हमारे Music Store Database में यदि हम CustID Foreign Key को Null Assign करते हैं, तो गम्भीर समस्याएं पैदा हो सकती हैं, क्योंकि यदि Orders Table में CustID Foreign Key को Null Assign किया जाता है, तो ये पता लगाने का कोई तरीका नहीं बचता है, कि उस Order को किस Customer ने Place किया है।

Primary Keys व **Foreign Keys** की Matching के आधार पर ही एक Relational DBMS विभिन्न प्रकार की Relationships को Represent करता है। उदाहरण के लिए मानलो कि Music Store का कोई Employee ये जानना चाहता है कि Order Number #600000 पर किस **Title** का Order किया गया है।

इस स्थिति में DBMS **Line Items** Table में सबसे पहले उन Rows को Identify करता है, जिनमें Order Number #600000 Stored है। फिर DBMS उन Identified Rows में से **Item Numbers** को Select करता है और उन्हें **Items** Table के **Item Numbers** से Match करता

है। जिस Row में दोनों **Item Numbers** Match होते हैं, DBMS उस Row से Associated **Title** को Retrieve कर लेता है।

Referential Integrity

पिछले Paragraph में Data के Access होने का जो तरीका बताया गया है, वह तरीका तब तक अच्छे तरीके से काम करता है, जब तक किसी कारणवश कोई ऐसा Record Orders Table में नहीं होता है, जिसमें OrderID Filed में Null हो। क्योंकि यदि Order Table में OrderID Field में Null Stored हो, तो उस Row से Match होने वाला कोई Record Order Lines Table में प्राप्त नहीं होगा।

ये एक बहुत ही अवांछित स्थिति होती है, क्योंकि इस स्थिति में Order किए गए Items को Ship नहीं किया जा सकता है, क्योंकि इस स्थिति में ऐसा कोई तरीका नहीं होता है, जिससे ये पता लगाया जा सके, कि उस Order को किस Customer ने Place किया है।

इसलिए इस प्रकार की स्थितियों से बचने के लिए Relational Data Model ***Referential Integrity*** नाम के एक Constraint को Enforce करता है, जो ये तय करता है कि हर Non-Null Foreign Key Value किसी Existing Primary Key Value से जरूर Match हो। किसी Relational Database में Use किए जाने वाले सभी Constraint की तुलना में ये सबसे ज्यादा महत्वपूर्ण Constraint होता है, क्योंकि ये Database के विभिन्न Entities के बीच के Cross-Reference की विश्वसनीयता या Constancy को सुनिश्चित या Ensure करता है।

Referential Integrity Constraints Database में Stored होते हैं और इन्हें DBMS द्वारा Enforce किया जाता है। अन्य Constraints की तरह ही, User जब भी Database में Stored किसी Data को Modify करने की कोशिश करता है या Database में नया Data Enter करता है, DBMS इस Constraint को Check करता है और इस बात का Verification करता है कि उस Data से सम्बंधित सभी Entities आपस में पूरी तरह से Compatible हैं। यानी हर **Foreign Key** उसके **Primary Key** से Matched है।

यदि इस Input किए जाने वाले या Modify किए जाने वाले Data द्वारा इस Constraint को Violet किया जाता है, तो DBMS उस Data Modification या Insertion को Allow नहीं करता और Database में स्थित किसी भी Entity के किसी भी Data को किसी भी प्रकार से Change नहीं करने देता है।

हमेंशा ये जरूरी नहीं होता है कि किसी Table की **Foreign Key** हमेंशा किसी दूसरी Table के **Primary Key** से ही Refer हो, बल्कि Foreign Key को केवल एक Primary Key के Reference की ही जरूरत होती है, इसलिए हम एक Foreign Key के साथ उसी Table की Primary Key का Reference भी Set कर सकते हैं, जिसमें वह Foreign Key Stored है। उदाहरण के लिए निम्न Employee Relation को देखिएः

Employee (**EmpID**, FName, LName, Dept, MngrID)

चूंकि Manager भी एक Employee ही होता है, इसलिए हालांकि इस Relation में **MngrID** को **EmpID** से अलग नाम दिया गया है, लेकिन फिर भी वास्तव में MngrID एक Foreign Key है, जो कि अपनी ही Table के Primary Key को Refer कर रहा है। इसलिए जितनी बार भी एक User एक MngrID Input करता है, DBMS हर बार इस बात को Ensure करता है कि वह Manager एक Employee की तरह उस Table में पहले से ही Exist है।

Views

वे लोग जो कि किसी Database Schema को Develop करने के लिए जिम्मेदार होते हैं, या वे लोग जो अन्य सामान्य Users के लिए Application Programs Develop करते हैं, वे सामान्यतया Database Schema व Database की Base Tables के साथ Directly काम कर सकते हैं।

लेकिन जो End Users होते हैं, उन्हें Application Programs या Database Schema के बारे में पूरी जानकारी नहीं होती है, इसलिए सामान्यतया इन्हें Database की Base Tables को Directly Use करने से रोका जाता है।

चूंकि End User को कभी भी Database Schema व Application Program के बारे में पूरी जानकारी नहीं होती है, इसलिए यदि वे Database की Base Tables के साथ Directly काम करते हैं, तो हो सकता है कि वे अपने Database की Tables में Stored Data को गलती से Corrupt कर दें। इसलिए Data की सुरक्षा के लिए End Users को Database की Base Tables के साथ Directly काम करने का अधिकार Database Developer द्वारा नहीं होता है।

इसलिए Relational Data Model एक ऐसा तरीका Provide करता है, जिसमें हर User को Database में उसका स्वयं का एक Window मिलता है और हर End User उसी Window में अपना काम करता है। ये Window Database Design की सभी Details को User के लिए Hide कर देता है, जिससे एक End User कभी भी Database की Base Tables को Direct Access नहीं कर पाता है और Database का Data End User की गलती से Damage होने से हमेंशा बचा रहता है।

View भी एक तरह का Table ही होता है, लेकिन ये Data के साथ Database में Store नहीं होता है। बल्कि ये एक नाम के साथ Data Dictionary में Store होता है। इसमें हमेंशा कोई Database Query होती है, जिसके आधार पर कोई View Database से अपना Data Retrieve करता है। एक View में एक से ज्यादा Tables, Rows व Columns के Data हो सकते हैं।

हालांकि एक View को किसी भी Database Query के आधार पर Create किया जा सकता है, लेकिन कई Views को केवल Data Display करने के लिए ही Create किया जाता है। किसी भी Database के Data को Modify करने के लिए Views को Create नहीं किया जाता है।

Views को इस तरीके से Store करने का सबसे बडा फायदा ये है कि जब भी User किसी Data Manipulation Language (SQL Statement) में View का नाम Include करता है, तब DBMS उस View Name से Associated Query को Execute करता है और View के Table को फिर से Recreate करता है। इसका मतलब ये है कि View में हमेंशा Current Data ही होता है।

एक View Table Main Memory में तभी तक उपलब्ध रहता है, जब तक कोई Data Manipulation Language (SQL Statement) Execute होता है। जैसे ही User दूसरा SQL Statement Use करके दूसरी Query Create करता है, पिछली Query की View Table Main Memory से Remove हो जाती है और Generate होने वाला नया Result View में Stored उस पिछली Query के Result को Replace कर देता है। इसलिए एक View Table हमेंशा एक **Virtual Table** होती है।

कुछ DBMS ये सुविधा देते हैं कि View Table के Contents को Base Table की तरह Store किया जा सकता है। लेकिन किसी View Table को Base Table बनाने का कोई विशेष Views को इस तरीके से Store करने का सबसे बडा फायदा ये है कि जब भी User किसी Data Manipulation Language (SQL Statement) में View का नाम Include करता है, तब DBMS उस View Name से Associated Query को Execute करता है और View के Table को फिर से Recreate करता है। इसका मतलब ये है कि View में हमेंशा Current Data ही होता है।

एक औचित्य नहीं होता है, क्योकि ऐसे DBMS Softwares में भी किसी View Table से Create की गई Base Table को उस स्थिति में Automatically Update करने की सुविधा नहीं होती है, जब उस Table में Change किया जाता है, जिसके आधार पर View Table को बनाया गया था।

यानी यदि किसी Table X से एक View Table Y को Create किया जाता है और इस View Table को Base Table के रूप में Database में Store कर लिया जाता है और उसके बाद Table X में कोई परिवर्तन किया जाता है, तो इस View Table Y से Create होने वाली Base Table Y में कोई Automatic Updation नहीं होता है।

इसलिए जैसे ही हम Base Table X में कोई परिवर्तन करते हैं, Base Table Y के Data हमारे Database के लिए **Out Of Date** हो जाते हैं, जिनका हमारे Database के लिए कोई Meaning नहीं रह जाता है।

अब एक सवाल दिमाग में आ सकता है कि जब Data को Store करने के लिए एक Base Table Create किया ही जाता है, तो Views को Use करने की क्या जरूरत है। तो Views को Use करने के तीन अच्छे कारण हैं:

1 जैसाकि पहले बताया गया कि Views का प्रयोग करने से Data की Security का Feature प्राप्त होता है, क्योंकि Views का प्रयोग करके End User कभी भी Database के Schema के साथ किसी प्रकार की कोई प्रक्रिया नहीं कर सकता है।

2 Views उन लोगों के लिए Database का Design समझना सरल बना देता है, जो किसी Database Schema के आधार पर Application Programs Develop करते हैं।

3 चूंकि Views एक ऐसी Query होती हैं, जिनका एक नाम होता है, इसलिए इनका प्रयोग ऐसी Complex Queries को Store करने के लिए किया जा सकता है, जिन्हें बार–बार Use करना होता है। फिर उन Complex Queries को Use करने के लिए हमें केवल उस View के नाम को ही Use करना होता है।

Relational Database के अन्य Structural Elements की तरह ही Views को भी किसी भी समय Create व Destroy किया जा सकता है। चूंकि Views में किसी प्रकार का Stored Data नहीं होता है, बल्कि किसी ऐसी Query का Specification होता है, जिसके आधार पर Virtual Table Create होती है, इसलिए Views को Add करने या Delete करने पर इसका Database की Base Tables व Tables में Stored Data पर कोई प्रभाव नहीं पडता है।

किसी View को Remove करने पर केवल तभी समस्या पैदा हो सकती है, जब उस View को किसी Application Program में Use किया गया हो और Program को इस प्रकार से Modify ना किया गया हो कि वह उस View के बिना भी किसी अन्य View या Base Table के आधार पर ठीक तरीके से काम कर सके।

Data Dictionary

हर DBMS में एक Data Dictionary या Catalog होता है। Relational Database का Structure इसी Catalog या Data Dictionary में Store होता है। Data Dictionary Relations के समूह से बनी होती है और हमारे Database के सभी Elements इस Data Dictionary Relation में उसी तरह से Store होते हैं, जिस तरह से एक Entity के Relation में विभिन्न Data Store होते हैं। यानी Data Dictionary Relation में Database के सभी Relations Data की तरह Store होते हैं।

हम Data Dictionary Relation की भी उन्हीं Tools का प्रयोग करके Query कर सकते हैं, जिन Tools का प्रयोग करके किसी सामान्य Relation की Query करते हैं। कोई भी User Data Dictionary Tables को Directly Modify नहीं कर सकता है।

फिर भी जिन Data Manipulation Language Commands (SQL Commands) का प्रयोग हम Database के Elements को Create व Destroy करने के लिए करते हैं, उन्हीं Commands का प्रयोग हम Data Dictionary Tables के Rows को Modify करने के लिए भी कर सकते हैं। हमें Data Dictionary में निम्न प्रकार की Information प्राप्त होती हैः

1. हर Table को Create करने वाले Columns की Definitions की Information
2. Relations पर Apply किए गए Integrity Constraints की Information
3. Security Information जो ये बताती है कि कौनसा User किस Table पर किस प्रकार के Operations को Perform कर सकता है।
4. Database Structure Elements जैसे कि View व अन्य User Defined Domains की Information

User जब भी किसी भी तरीके से Data को Access करने की कोशिश करता है, Relational DBMS सबसे पहले Data Dictionary में ये तय करने के लिए जाता है कि User ने जिस Database Elements की Request की है, क्या वास्तव में वे Elements Database Schema का हिस्सा हैं अथवा नहीं। साथ ही DBMS ये भी Verify करता है कि क्या User को उस जानकारी को प्राप्त करने का अधिकार है, जिसके लिए उसने Request किया है।

जब User Database के किसी Data को Modify करना चाहता है, तब भी DBMS Data Dictionary में जाता है और उन Integrity Constraints को Check करता है, जिन्हें उन Relation पर Place किया गया हो सकता है। यदि DBMS का Data के साथ Constraint Match हो जाता है, और DBMS को लगता है कि User एक Legal Operation कर रहा है, तो DBMS उस User की Request को पूरा करता है और Database के किसी Relation के Data को Modify करने की Permission दे देता है।

लेकिन यदि User Data के साथ Apply किए गए Constraint की जरूरत को पूरा नहीं करता है, तो DBMS User को एक Error Message देता है और Database के किसी भी Relation के किसी भी Data में कोई Change नहीं करता है।

क्योंकि Relational Database में सभी प्रकार के Data Accesses Data Dictionary के Through होते हैं, इसलिए Relational DBMS को **Data Dictionary Driven** भी कहा जाता है। वे सभी Relations जो एक Data Dictionary को Create करते हैं, कुछ हद तक DBMS पर निर्भर होते हैं। इसलिए विभिन्न प्रकार के DBMS विभिन्न तरीकों से इस Data Dictionary को Manage करते हैं।

DBMS

THE NORMALIZATION

DBMS – The Normalization

विभिन्न प्रकार के Entities व उनके Attributes को कई तरीकों से किसी Relation के रूप में Represent किया जा सकता है। इस अध्याय में हम Normalization के Process को समझेंगे। जब हम इस Process के आधार पर विभिन्न Relations Create करते हैं, तब एक खराब Database Design से पैदा होने वाली विभिन्न प्रकार की समस्याएं Avoid हो जाती हैं।

Database के Normalization के दो तरीके प्रचलित हैं। पहले तरीके में एक ER Diagram के आधार पर Normalization किया जाता है। इस तरीके में यदि ER Diagram को Correctly Draw किया गया है, तो हम कुछ Simple Rules को Follow करते हुए उस ER Diagram को ऐसे Relations में Translate कर सकते हैं, जो ज्यादातर Relational Design Problems को Avoid हो जाता है।

इस Normalization Process की समस्या ये है कि इस तरीके के आधार पर जो Database Design बनता है, वह Design सही है या नहीं, इस बात को निश्चित करने का कोई तरीका नहीं होता है। दूसरे तरीके में हम विभिन्न Relations Create करने के लिए Theoretical Concept को Use करते हैं। ये तरीका पहले तरीके की तुलना में थोडा अधिक जटिल है, लेकिन इससे बनने वाला Design एक Better Design होता है।

Practically इन दोनों तरीकों के Combination को Use करके, ज्यादा आसानी से एक अच्छा Design Create कर सकते हैं। सबसे पहले हम ER Diagram Create करते हैं और इसका प्रयोग करके Relations Create करते हैं। उसके बाद दूसरे तरीके के Theoretical Rules को उन Relations पर Apply करके Design को Check करते हैं।

Translating an ER Diagram into Relations

एक ऐसा ER Diagram, जिसके सभी **Many To Many** Relationships को Composite Entities का प्रयोग करके **One To Many** Relationships में Convert कर लिया गया हो, तो Directly Database Relations में Translate कर सकते हैं। ऐसा करने के लिए हमें निम्न Steps को Follow करने होते हैं:

1. हर Entity के लिए एक Table Create करते हैं।
2. हर वह Entity जो किसी एक या एक से ज्यादा Relationships के केवल "**One**" End की तरफ हो और "**Many**" End की तरफ ना हो, ऐसे Entity की Table में केवल एक **Single-Column Primary Key** को Define करना होता है।
3. हर वह Entity जो किसी एक या एक से अधिक Relationship के "**Many**" End की तरफ हो, ऐसे Entity की Table में उसके Parent Table, जो कि "**One**" End की तरफ होता है, की **Primary Key** को अपनी Table में **Foreign Key** की तरह Use करना चाहिए।

4 यदि एक Entity जो किसी एक या एक से ज्यादा Relationships के "**Many**" End की तरफ हो और उसमें कोई Natural Primary Key हो, जैसे कि Invoice Number या Order Number, तो इस Single-Column Primary Key को Use करना चाहिए। लेकिन यदि ऐसा ना हो, तो इस Table की Parent Table के Primary Key को किसी अन्य Column या Columns के Group के साथ Composite Primary Key के रूप में Use करना चाहिए।

इन Guidelines को Follow करके हम "**Music Store**" Database के Design को निम्नानुसार Theoretically Represent कर सकते हैं:

Customer (CustID, FName, LName, Street, City, State, Pincode, Telephone, CreditCardNo, CardExpiryDate)

Item (ItemID, Title, DistID, RetailPrice, ReleaseDate, Genre)

Order (OrderID, CustID, OrderDate, OrderFilled)

OrderLines (OrderID, ItemID, Quantity, DiscountApplied, SellingPrice, LineCost, Shipped)

Distributor (DistID, Name, Street, City, State, Pincode, Telephone, ContactPerson, ContactPersonExt)

Actor (ActorID, Name)

Performance (ActorID, ItemID, Role)

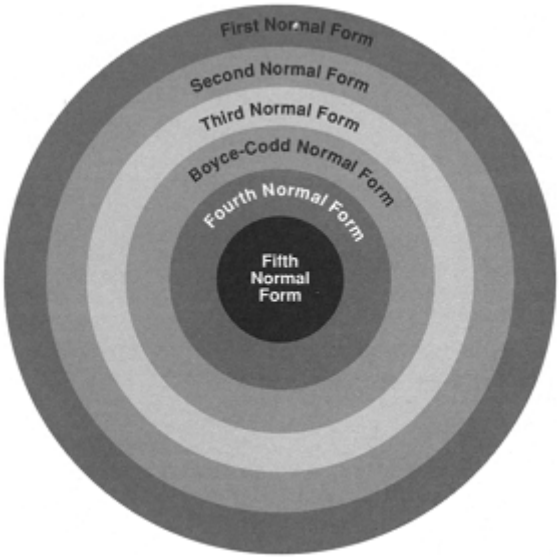
Producer (ProducerID, Studio)

Production (ProductionID, ItemID)

इन Relations को थोडा सा Modify किया गया है, लेकिन इन Modifications का ER Diagram या Database के काम करने के तरीके पर कोई अन्तर नहीं पडा है।

Normal Forms

वे Theoretical Rules जिनका किसी भी Relational Database Design के Compatible होना जरूरी होता है, Normal Forms कहलाते हैं। हर Normal Forms कुछ Strict Rules का समूह होता है। Theoretically Database जितने Higher Form में होता है, Relations के बीच का Design उतना ही अच्छा होता है।



जैसाकि हम पिछले चित्र में देखकर समझ सकते हैं, किसी भी Design की 6 Normal Forms हो सकती हैं। ये चित्र इस बात को दर्शाता है कि यदि कोई Design किसी Higher Form में है तो वह Design उसकी सभी Inner Normal Form में जरूर होता है। यानी यदि कोई Design Inner Normal Form में नहीं हो तो उसे Higher Normal Form में Define नहीं किया जा सकता है।

ज्यादातर Cases में यदि हम हमारे Relations या Tables Third Normal Form (3NF) तक भी Normalize कर लेते हैं, तो हम खराब Database Design के ज्यादातर Problems को Solve कर चुके होते हैं। यानी किसी Database की सभी Tables को 3rd Normal Form तक Normalize कर लेने पर उस Database की Design सम्बंधित ज्यादातर परेशानियां समाप्त हो जाती हैं।

Boyce-Codd (BCNF) व 4th Normal Form को विशेष Situations को Handle करने के लिए कभी–कभार ही Database पर Apply किया जाता है। हालांकि इन दोनों Normalization Processes को भी Conceptually समझना सरल होता है और जरूरत पडने पर इन्हें Practice में भी लिया जा सकता है।

Fifth Normal Form के नियम बहुत ही Complex होते हैं और इन्हें Practically Use करना काफी मुश्किल होता है। कोई Relation 5th Normal Form में है या नहीं, इस बात को Verify करना काफी मुश्किल होता है। ज्यादातर Database Designers 5th Normal Form तक किसी भी Relation को Normalize नहीं करते हैं। यदि उन्हें लगता है कि उनका Relation 3rd Normal Form या जरूरत के आधार पर 4th Normal Form में है, तो वे ये मान लेते हैं कि उनका Design Generally Problem Free है।

चित्र में दर्शाए गए 6 Normal Forms के अलावा एक और Normal Form होता है, जिसे **Domain/Key Normal Form** कहा जाता है। ये एक पूरी तरह से Theoretical Concept होता है और आज के समय में इस Normal Form को किसी भी Practical Design में Use नहीं किया जाता है।

First Normal Form

यदि किसी Table के सारे Data एक 2-Dimensional Table के रूप में हैं और उनमें से किसी भी Data के Group का Repetition नहीं हो रहा है, तो उस Table को 1st Normal Form में माना सकते है।

1st Normal Form को समझने का मुख्य आधार ये है कि हम Data के Repeating Group की प्रकृति को समझें। किसी Relation का एक ऐसा Attribute, जिसमें एक से ज्यादा Data Store हो सकते हों, को **Repeating Group Data** कहा जाता है। जब हम किसी Repeating Group Data को ER Diagram में Represent करना चाहते हैं, तब हम उस Repeating Group Data को Multi-Valued Attribute के रूप में Represent करते हैं।

उदाहरण के लिए मानलो कि हम किसी Employee के Relation के साथ प्रक्रिया कर रहे हैं और हमें किसी Employee के बच्चों के नाम व Birth Date को Employee के Relation में Data के रूप में Store करना है।

अब चूंकि एक Employee के एक से ज्यादा बच्चे भी हो सकते हैं, इसलिए एक ही Employee को Represent करने वाले एक ही Row के **Child Name Filed** व **Child Birth Date Field** में एक से ज्यादा मानों को Store करने की जरूरत पड सकती है, जबकि किसी भी Relational Database में किसी एक Row के एक Field में केवल एक ही मान को Store किया जा सकता है।

इस स्थिति में Child का नाम व Birth Date Employee के Record में एक से ज्यादा बार Repeat हो सकते हैं, इसलिए इन दोनों Data को Employee के Relation के लिए **Repeating Group of Data** कहा जा सकता है। इस बात को हम एक सारणी द्वारा निम्नानुसार दर्शा सकते हैं:

EmpID	FName	LName	DOB	ChildName	ChildBirthDate
001	Rahul	Sharma	10/12/1982	**Rohit Sharma**	**15/12/2006**
				Mohit Sharma	**20/10/2008**

इस Table में हम देख सकते हैं कि Rahul नाम के एक Employee के दो बच्चे Rohit व Mohit हैं, लेकिन हम दोनों बच्चों के Data को Employee के Relation में Store नहीं कर सकते हैं, क्योंकि यदि हम ऐसा करने की कोशिश करते हैं, तो हमें एक ही Attribute Column में एक से

ज्यादा नामों व Birthdates को Store करना होगा, जो कि एक Relational Database में सम्भव नहीं है।

किसी Relational Database में Repeating Groups को Allow ना करने का एक अच्छा कारण भी है। इस कारण को समझने के लिए हम एक उदाहरण देखते हैं, जिसमें एक Table में निम्नानुसार कुछ Employees व उनके बच्चों के नाम Stored हैं:

EmpID	FName	LName	Children's Names	Children's Birthdates
1001	Jane	Doe	Mary, Sam	1/1/92, 5/15/94
1002	John	Doe	Mary, Sam	1/1/92, 5/15/94
1003	Jane	Smith	John, Pat, Lee, Mary	10/5/94, 10/12/90, 6/6/96, 8/21/94
1004	John	Smith	Michael	7/4/96
1005	Jane	Jones	Edward, Martha	10/21/95, 10/15/89

इस Table में हम देख सकते हैं कि एक ही **Single Row** के **Children Name Field** व **Children Birthdates Field** दोनों ही Fields में एक से ज्यादा मान Stored हैं। ये व्यवस्था दो बडी Problems Create करती है:

1. इस Table में ऐसा कोई तरीका नहीं है, जिससे ये जाना जा सके कि किस Child का Birth Date क्या है। इस व्यवस्था को Use करने पर हमें हमेंशा Children के नाम व Birth Date दोनों को समान क्रम में Store करना जरूरी होता है। किसी Table में Children Name Field में जिस बच्चे का नाम पहले लिखा गया है, हमें उसी बच्चे का Birth Date पहले लिखना होता है और ऐसा कोई निश्चित तरीका नहीं होता है, जिससे ये Verify हो सके कि हमने जिस बच्चे का नाम पहले लिखा है, हम उसी बच्चे की Date Of Birth भी पहले ही लिख रहे हैं या नहीं।

2. जब हम किसी Table में Multi-Valued Data Store करते हैं, तब DBMS किसी Data को Search करने के लिए उस Multi-Valued Column को Extract करता है और उस Column पर Sequential Processing को Apply करके Required Data को प्राप्त करता है, जो कि सबसे धीमी Searching Process होती है।

इन दोनों समस्याओं का एक ही Solution है कि किसी भी Single Column में Multiple Values को Store ना किया जाए यानी **Repeating Group of Data** को किसी भी Relation में Avoid किया जाए। Repeating Groups की समस्या से बचने और Relation को First Normal Form में लाने के दो तरीके हैं। पहला तरीका एक सही तरीका है जबकि दूसरा तरीका एक गलत तरीका है।

हम पहले गलत तरीके को देखेंगे ताकि हम ये समझ सकें कि हमें एक Database में क्या नहीं करना चाहिए। इस गलत तरीके को हम निम्न सारणी द्वारा समझ सकते हैं, जिसमें किसी

Repeating Group के Data की विभिन्न Values को Handle करने के लिए Relation में Columns की संख्या को Increase कर लिया जाता है:

EmpID	FName	LName	Child1	DOB1	Child2	DOB2	Child3	DOB3
1001	Jane	Doe	Mary,	1/1/92	Sam	5/15/94		
1002	John	Doe	Mary	1/1/92	Sam	5/15/94		
1003	Jane	Smith	John	10/5/94	Pat	10/12/90	Lee	6/6/96
1004	John	Smith	Michael	7/4/96				
1005	Jane	Jones	Edward	10/21/95	Martha	10/15/89		

इस Example में किसी Employee के तीन Childs के नाम व Birth Dates को Store किया जा सकता है। ये Table First Normal Form के Criteria से मेल नहीं कर रहा है। हालांकि इस Table में Data के Repeating Groups नहीं हैं और हर Child की Birth Date को अलग Store किया गया है। फिर भी इस Design में कई Problems हैं, जो निम्नानुसार हैं:

1 इस Relation में हम सिर्फ तीन Child के ही Name व Birth Date को Store कर सकते हैं। इसलिए यदि हम Jane Smith के चौथे बच्चे का नाम व Date Of Birth Store करना चाहें, तो इस Relation में उस Child के लिए कोई जगह नहीं है। यदि हम चौथे बच्चे का भी Data इसी Relation में Store करना चाहें, तो या तो हमें एक और Field Pair Create करना होगा या फिर एक और Row में इस Data को Store करना होगा।

   यदि हम चौथे बच्चे के Data को एक और Row में Store करते हैं, तो फिर उस बच्चे को उसके Father के EmpID से Relate करने के लिए हमें EmpID के साथ कम से कम एक Child के नाम के Column को भी मिलाना होगा। यानी हमें Composite Key का प्रयोग करना होगा।

2 यदि हम इस तरीके को Use करते हैं, तो जिन Employees के एक या दो ही Child हैं, उनके Row में तीसरे Child के Data की जगह Empty रहेगी, जिससे Employee के Record द्वारा Reserved Space Waste होगा।

3 इस तरीके को Use करने पर किसी Particular Child की Searching करना काफी मुश्किल हो जाता है। इस Design में यदि DBMS से ये पूछा जाए, कि **"क्या इस Relation में Lee नाम का कोई Child है या नहीं?"** तो DBMS को एक Query Construct करना पडता है, जिसमें तीनों Childs के नाम Included होते हैं, क्योंकि DBMS के पास ऐसा कोई तरीका नहीं होता है, जिससे वह Particular "Lee" के ही Column को Identify कर सके।

Repeating Group को Handle करने का सही तरीका ये है कि Repeating Group के Data को Store करने के लिए एक और Entity यानी Table Create किया जाए, जो Repeating

Group के Multiple Records या Instances को Handle कर सके। यदि हम हमारे इस उदाहरण के सन्दर्भ में देखें, तो हमें Children नाम की एक और Table को Create करना होगा, जिसमें निम्नानुसार Data Store किए जा सकते हैं:

Employees

EmpID	FirstName	LastName
1001	Jane	Doe
1002	John	Doe
1003	Jane	Smith
1004	John	Smith
1005	Jane	Jones

Employees

EmpID	ChildName	BirthDate
1001	Mary	1/1/92
1001	Sam	5/15/94
1002	Mary	1/1/92
1002	Sam	5/15/94
1003	John	10/5/94
1003	Pat	10/12/90
1003	Lee	6/6/96
1003	Mary	8/21/94
1004	Michael	7/4/96
1005	Edward	10/21/95
1005	Martha	10/15/89

हम देख सकते हैं कि अब दोनों ही Tables में कोई भी Repeating Group Of Data नहीं है इसलिए इस Design से पहले बताई गई सभी प्रकार की समस्याएं समाप्त हो जाती हैं। यानी इस Design से हमें उपरोक्त सभी Problems का निम्नानुसार Solution प्राप्त हो जाता है:

- इस Design में हम किसी Employee के सभी Childs के Name व Birth Dates को Store कर सकते हैं, क्योंकि यहां पर Store किए जाने वाले Children पर किसी प्रकार की कोई Limitation नहीं है।
- इस Design में उतना ही Space Use होता है, जितने की जरूरत विभिन्न Childs के Data को Store करने के लिए होती है, इसलिए Storage Space Waste नहीं होता है।
- इस Design में किसी Child की Searching करना काफी सरल हो जाता है, क्योंकि Child का नाम केवल एक ही Column में Store हो सकता है, इसलिए DBMS को किसी

भी Child का नाम Search करने के लिए केवल एक ही Column को ध्यान में रखना होता है।

हालांकि First Normal Form के Relations में Data के Repeating Groups नहीं होते हैं, लेकिन उनमें अन्य प्रकार की कई और Problems होती हैं। First Normal Form के Relation की समस्याओं को समझने के लिए हम Music Store Database के Data Entry Form से Connected Back-End Table को ही उदाहरण के रूप में ले रहे हैं, जिसमें Data Entry Form से Enter किया गया Data जाकर Store होता है। **Music Store** Data Entry Form से Connected Back-End Relation के Notation को हम निम्नानुसार Represent कर सकते हैं:

Orders (CustID, FName, LName, Street, City, State, Pincode, Telephone, OrderDate, ItemID, Title, Price, HasShipped)

इस Table में हमें जो सबसे पहले काम के रूप में Primary Key को Determine करना होता है। इस Table में केवल Customer Number से किसी Order को पूरी तरह से Uniquely Identify नहीं किया जा सकता है, क्योंकि हर Item के Order के साथ ही CustID Repeat होता है।

इसी तरह से केवल ItemID को Primary Key के रूप में Use नहीं किया जा सकता है, क्योंकि हर Order के साथ ये भी Repeat होता है। इस Relation में Primary Key का केवल एक ही उचित तरीका हो सकता है और वह तरीका Composite Key का है।

इस उदाहरण में हम OrderID व ItemID दोनों के Combination को Primary Key के रूप में Use कर सकते हैं। यदि हम OrderID व ItemID के Combination को Primary Key के रूप में Use करते हैं, तो इस Relation में हम दो बहुत ही महत्वपूर्ण कामों को पूरा नहीं कर सकते हैं, जिन्हें निम्नानुसार बताया गया है:

1. हम तब तक किसी Customer के Data को Relation में Store नहीं कर सकते हैं, जब तक कि वह Customer कम से कम एक Order Place ना करे, क्योंकि बिना एक **Order** और उस Order पर स्थित एक **Item** के, हमारे पास एक Complete Primary Key नहीं होती है।

2. इसी तरह से हम तब तक किसी Item की Information को भी Relation में Store नहीं कर सकते हैं, जब तक कि कोई Customer किसी Order द्वारा उस Item को Order ना करे, क्योंकि बिना **OrderID** के भी एक Complete Primary Key नहीं बन सकता।

ये दोनों कारण **Insertion Anomalies** हैं, जो एक ऐसी Situation को Represent कर रहे हैं, जिसमें हम किसी Relation में इसलिए किसी Data को Insert नहीं कर सकते हैं, क्योंकि हमारे पास एक Complete Primary Key नहीं है।

कोई भी Relation, जो कि First Normal Form में होता है, उसमें Insertion Anomalies की स्थिति Common रूप से होती ही है। Insertion Anomalies की स्थिति तब पैदा होती है, जब किसी एक Relation में एक से ज्यादा Entities के Data को Store करने की कोशिश की जाती है। इस Situation में Anomaly हमें उस समय एक Unrelated Entity जैसे कि **Item** के Data को Insert करने के लिए मजबूर करता है, जब हम किसी दूसरे Entity जैसे कि **Customer** के Data को Insert कर रहे होते हैं।

First Normal Form के Relations उस समय भी समस्याएं पैदा करते हैं, जब हम उस Relation से किसी Data को Delete करने की कोशिश करते हैं। उदाहरण के लिए मानलो कि हम उस Customer के Order को Delete करना चाहते हैं, जिसने अपने Single Item का Order Cancel कर दिया है। अब यदि

1. Customer ने पहली ही बार Order दिया हो और केवल एक ही Item का Order दिया हो, तो उस Customer के Order को Delete करने पर उस Customer की Information भी Music Store के Database से Permanently Delete हो जाएगी, जिससे **Music Store Organization** का उस Customer से तब तक के लिए Permanently Link टूट जाएगा, जब तक कि वह दुबारा कोई Order नहीं देता है।

2. Customer ने पहली ही बार उस Item का Order दिया हो और उस Customer से पहले किसी दूसरे Customer ने उस Item का Order नहीं दिया हो, तो उस Order को Delete करने पर उस Item की Information भी **Music Store** के Database से Permanently Delete हो जाएगी।

3. Customer ने अपने Order में केवल एक ही Item का Order दिया हो, तो Order को Delete करने पर उस Order की पूरी Information Database से Delete हो जाएगी।

ये **Deletion Anomalies** इसलिए पैदा होती हैं, क्योंकि किसी Row की Primary Key के एक Part में उस समय **Null** Store हो जाता है, जब Item के Data को Delete करते समय हमें Data की पूरी Row को Delete करना पडता है। Deletion Anomaly का परिणाम ये होता है कि Music Store Database से वे Data भी Delete हो जाते हैं, जिन्हें हम Database में Stored रखना चाहते हैं।

यदि हम Practical रूप से देखें तो जब हमें किसी अवांछित Entity के Data को Delete करना होता है, तब इस Anomaly की वजह से हमें उस Entity के Data को भी Delete करना पडता है, जो कि Same Table में तो होते हैं लेकिन अवांछित Entity से Unrelated होते हैं। इस पूरे Discussion का सारांश ये है कि एक ही Table में एक से ज्यादा Entities के Data को Store नहीं करना चाहिए।

Orders के Relation में एक अन्तिम Anomaly **Update/Modification Anomaly** भी है। Orders Relation में हर Order के साथ हर Customer की Information को बार–बार Store करने की वजह से **Music Store** Database में एक ही Customer के बहुत सारे Unnecessary Duplicated Data Store हो जाते हैं।

इसलिए जब एक Customer Move होता है, तब उस Customer ने जितने भी Items के जितने भी Orders Music Store Organization को दिए होते हैं, Database में उन सभी Items के Orders की Row को Modify करके Customer के Data को Change करना पडता है, क्योंकि हर Item की Entry Database में करने के लिए उस Item की Information के साथ Customer की Information को भी Database में Store किया जाता है।

अब यदि हर Row को Correctly Change ना किया जाए, तो किसी Particular Customer की Information को Represent करने वाले जिन सभी Data को हमेंशा समान होना चाहिए, वे समान नहीं रह जाते हैं। Data की इस Inconsistency की सम्भावना के कारण **Modification Anomaly** की Situation पैदा होती है।

Second Normal Form

First Normal Form की विभिन्न Anomalies को हटाने का समाधान ये है कि First Normal Form वाली Relation से सभी Entities को एक अलग Relation के रूप में Define किया जाए। उदाहरण के लिए Music Store के इस Orders Relation में से हम चार स्वतंत्र Entities (**Customers, Items, Orders** व **Line Items**) को अलग कर सकते हैं। ऐसा करने पर **Music Store Organization** का ये Relation **Second Normal Form** में आ जाता है। Theoretical शब्दों में Second Normal Form को निम्नानुसार परिभाषित किया जा सकता हैः

जब Relation First Normal Form में हो और सभी Non-Key Attributes, Functionally सिर्फ Primary Key पर Dependent हो। यदि कोई Non-Key Attribute Functionally केवल Primary Key पर Depend ना होकर किसी Non-Key Attribute पर Depend हो, तो उस Non-Key Attribute और उस पर Depend सभी अन्य Non-Key Attributes को उस Relation से हटाकर एक नए Relation में Define करना चाहिए और इस नए Relation में उस Key को Primary Key बना देना चाहिए, जिस पर अन्य Attributes Depend हों।

Functional Dependency दो Attributes के बीच की एक One-Way Relationship होती है। जैसे किसी Relation में किसी भी समय एक Attribute **A** से किसी दूसरे Attribute **B** की केवल एक ही Value Associated होनी चाहिए।

उदाहरण के लिए मानलो कि **Orders** Relation में **A** एक Customer का Customer Number या **CustID** है। अब हर Customer का Customer Number एक First Name, एक Last

Name, एक Street Address, एक City, एक State, एक Pincode व एक Telephone Number से Associated होता है।

हालांकि इन Attributes की Values को किसी भी समय Change किया जा सकता है, लेकिन किसी भी समय हर Attribute में केवल एक ही मान होता है। इस स्थिति में हम कह सकते हैं कि First Name, Last Name, Street Address, City State, Pincode व Telephone Numbers ये सभी Functionally Customer Number पर Dependent हैं। Attributes के बीच की इस Relationship को अक्सर निम्नानुसार Represent किया जाता हैः

```
CustID -> FName, LName, Street Address, City, State, Pincode, Telephone
```

और इसे इस तरह Read किया जाता है कि "Customer Number Determines First Name, Last Name, Street Address, City State, Pincode and Telephone Numbers". इस Relationship में Customer Number यानी **CustID** को ***Determinant*** के रूप के जाना जाता है, जो कि एक ऐसा Attribute होता है, जो अन्य Attributes की Values को Determine करता है।

ध्यान रखें कि Functional Dependency को Reverse Direction में Represent नहीं किया जा सकता है। उदाहरण के लिए किसी भी First Name या Last Name को एक से ज्यादा Customer Numbers के साथ Associate किया जा सकता है। **Orders** Table में निम्न Functional Dependencies हैंः

```
CustID -> FName, LName, Street Address, City, State, Pincode, Telephone
ItemID -> Title, Price
OrderID-> CustID, OrderDate
ItemID  + OrderID -> HasShipped
```

ध्यान दें कि Relation में हर Entity के लिए एक Determinant है और Determinant वही है, जिसे हमने Entity Identifier के रूप में Choose किया है। जब किसी Entity में Composite Identifier होता है, तब Determinate भी Composite होता है, जैसाकि चौथे Representation में **ItemID+OrderID** का Group एक Composite Identifier है। इस Example में कोई Order Ship किया जा चुका है अथवा नहीं, ये **ItemID** व **OrderID** के Combination पर Depend करता है।

जब हम किसी Database Environment में किसी Relation के विभिन्न Attributes के बीच की Functional Dependencies को Correctly Identify कर लेते हैं, उसके बाद हम इनका प्रयोग Relations को Second Normal Form में Transform करने के लिए कर सकते हैं।

इस स्थिति में हर Determinant Relation का Primary Key बन जाता है और जितने भी Attributes इस Determinant पर Depend होते हैं, वे सभी Attributes Relation के **Non-Key Attributes** बन जाते हैं। इस Concept के आधार पर Music Store Organization के Original Relation में से जिन चार Entities को स्वतंत्र रूप से Identify करके अलग किया जाता है, उन्हें निम्नानुसार Represent किया जा सकता है:

```
Customer (CustID, FName, LName, Street Address, City, State, Pincode, Telephone)
Items (ItemID, Title, Price)
Orders (OrderID, CustID, OrderDate)
LineItems (ItemID, OrderID, HasShipped )
```

ये चारों ही Relations ER Diagram के एक Single Entity से सम्बंधित होते हैं। ध्यान दें कि Database Design को **Functional Dependencies** व **Entities** दोनों में से किसके आधार पर Derive किया जाए, इसका कोई निश्चित नियम नहीं होता है।

महत्वपूर्ण बात ये होती है कि ER Diagram व अपने Relation में Identify की गई Functions Dependency दोनों के बीच Consistency होनी चाहिए। इस बात से Database के Design पर कोई प्रभाव नहीं पडता है कि हम अपने Relation को Functional Dependency के आधार पर Design करते हैं या Entities के आधार पर।

ज्यादातर स्थितियों में Database Design एक Interactive Process होता है, जिसमें हम Database का Initial Design Create करते हैं, उसे Check करते हैं, Modify करते हैं और फिर से Check करते हैं। हम Design Process के किसी भी Stage में Function Dependency और/या Entities को देख सकते हैं और एक दूसरे के Against Check कर सकते हैं।

क्योंकि हमेंशा ये जरूरी नहीं होता है कि हमने जिस Relation को First Normal Form में मान लिया है, वह वास्तव में First Normal Form में हो। Design Process के किसी भी Stage में हमें ऐसा महसूस हो सकता है, कि Relation पूरी तरह से First Normal Form में नहीं है और उसे फिर से First Normal Form में लाने की जरूरत है। जब हम Relation पर Second Normal Form के Criteria Rules को Apply करते हैं, तब Original Relation में Present Anomalies Eliminate हो जाती हैं और हम निम्न काम कर सकते हैं:

1. Customer के Order Place करने से पहले ही हम उस Customer के Data को Database Relation में Store कर सकते हैं।
2. हम किसी Order के Data को बिना Items की Information के भी Database Relation में Store कर सकते हैं।
3. किसी Customer द्वारा किसी Particular Item का Order दिए जाने से पहले भी हम Item के Data को Database Relation में Store कर सकते हैं।

4. अब Line Items को किसी भी Order से Delete किया जा सकता है। ऐसा करने पर Item को Describe करने वाले Data, स्वयं **Order** या किसी Item की Information पर इसका कोई प्रभाव नहीं पडता है।
5. Customer से सम्बंधित Data को केवल एक ही बार Store किया जाता है, इसलिए यदि Customer के Data में किसी प्रकार का Change करना पडे, तो ये Change केवल एक ही बार करना पडता है। इसमें Modification Anomaly का प्रभाव नहीं पडता है, क्योंकि Customer के Data को Database Relation में कई बार Store नहीं किया जाता है।

हालांकि Second Normal Form विभिन्न Relations में से ज्यादातर समस्याओं को समाप्त कर देता है। बहुत कम बार ही ऐसी स्थितियां होती हैं, जब हमारा Relation Second Normal Form में होता है, फिर भी उसमें Anomalies होती है।

उदाहरण के मानलो कि Music Store जिन Distributors से Titles लेता है, उन सभी Distributors के पास केवल एक ही Store Room है, जहां पर सिर्फ एक Telephone है। इस स्थिति में निम्न Relation Second Form में होगाः

```
Items (ItemID, Title, Distributor, WareHousePhoneNo)
```

हर ItemID के लिए इस Relation में केवल एक Title, एक Distributor व एक Warehouse Telephone Number है। इसलिए इस Relation में एक **Insertion Anomaly** है। हम तब तक किसी Distributor का Data Music Store Database में Store नहीं कर सकते हैं, जब तक कि हमें उस Distributor से कोई Item प्राप्त नहीं होता है।

साथ ही इस Relation में एक **Deletion Anomaly** भी है, क्योंकि यदि हम किसी Distributor द्वारा भेजे गए Only Item की Details को Delete कर देते हैं, तो हम Distributor की Information को भी खो देंगे।

इस Relation में हर Item के Record के साथ Distributor के Warehouse के Phone Number को भी Store किया जाता है, जिससे इसका बार–बार Duplication भी होता है, इसलिए इस Relation में **Modification Anomaly** भी है। इस स्थिति में ये Relation **Second Normal Form** में तो है, लेकिन **Third Normal Form** में नहीं है।

Third Normal Form

किसी Relation को Third Normal Form के आधार पर इसलिए Normalize किया जाता है, ताकि उपर बताई गई Anomalies का समाधान हो सके। यदि हम Entities के आधार पर देखें, तो Items Relation में **Item** व **Distributor** दो Entities से Related Data Store हो रहे हैं।

इसलिए उपरोक्त Anomalies को हटाने के लिए हमें इस Relation को निम्नानुसार दो Individual Relations में Divide करना होगाः

Items (ItemID, Distributor)
Distributors (DistID, WareHousePhoneNo)

Third Normal Form का Theoretical Definition ये है कि कोई Relation तब Third Normal Form में होता है, जब उस Relation में कोई **Transitive Dependencies** नहीं होती हैं। Original Relation में हमने जिस Functional Dependencies के बारे में जाना था, उसे ही **Transitive Dependency** कहते हैं। एक Relation में Transit Dependency तब Exist होती है, जब हमारे सामने निम्नानुसार Functional Dependency होती हैः

A -> B and B -> C So Indirectly A -> C

यही Dependency Original **Items** Relation में है। Warehouse के Phone Number का Functionally Item Number पर Depend होने का Only कारण यही है कि Distributor Functionally Item Number पर Dependent है और Phone Number Functionally Distributor पर Dependent है। इसलिए वास्तविक Functional Dependency निम्नानुसार हैः

ItemID -> Distributor
Distributor -> WareHousePhoneNo

जबकि **WareHousePhoneNo** Indirectly **ItemID** पर Dependent है, जिसे हम **Transitive Dependency** कहते हैं। Transitive Dependency को यदि हम Mathematical Example के रूप में Represent करें, तो यदि **A** Directly बडा हो **B** से और **B** Directly बडा हो **C** से तो **A** Indirectly **C** से भी बडा होता है।

हमारे Original **Items** Relation में दो Determinants हैं और दोनों ही Determinants को उनके Relation का **Primary Key** होना चाहिए, क्योंकि हर Determinate हमेंशा अपने Relation का Primary Key ही होता है।

हालांकि इस Relation में Second Determinate का एक Attribute के रूप में Exist होना ही Transitive Dependency का कारण नहीं है। बल्कि वास्तव में Transitive Dependency का मुख्य कारण ये है कि दूसरा Determinant Relation का **Candidate Key** नहीं है। इसे समझने के लिए निम्न Relation का उदाहरण देखते हैंः

Items (ItemID, UpcCode, Distributor, Price)

ItemID वह Number है, जिसे Music Store Organization अपने हर Item को Uniquely Identify करने के लिए Use करता है, जबकि UPC Code वह Industry-Wide Code है, जिसे हर Item को Uniquely Identify करने के लिए Use किया जाता है। अब इस Relation में Functional Dependencies निम्नानुसार हैं:

```
ItemID -> UpcCode, Distributor, Price
UpcCode -> ItemID, Distributor, Price
```

क्या इस Relation में अब Transitive Dependency है। नहीं, अब इन Relations में Transitive Dependencies नहीं हैं, क्योंकि Second Determinant एक **Candidate Key** है। क्योंकि "**Music Store**" UpcCode को भी उतनी ही आसानी से किसी Item को Uniquely Identify करने के लिए Use कर सकता है, जितनी आसानी से वह Primary Key को Use करता है। इस Relation में अब किसी प्रकार का कोई **Insertion, Deletion** या **Modification Anomaly** नहीं है और ये Relation अब केवल एक **Item** Entity को ही Describe कर रहा है।

Transitive Dependency किसी Relation में तब Exist होती है, जब कोई Determinant, जो कि Relation के लिए Primary Key नहीं होता है वह उस Relation का Candidate Key भी नहीं होता है।

उदाहरण के लिए हम जिस Items Table को उदाहरण के रूप में उपयोग में ले रहे हैं, उसमें Distributor एक Determinant है लेकिन वह Distributor उस Items Table के लिए Candidate Key नहीं है। क्योंकि एक Distributor से एक से ज्यादा Items **Music Store** में आ रहे हैं।

जब Second Normal Form के Relation में कोई Transitive Dependency होती है, तब हमें उस Relation को दो छोटे–छोटे Relations में Divide कर लेना चाहिए और दोनों Relations में दोनों Determinants को Primary Key बना लेना चाहिए। जिस Attributes को Determinants द्वारा Determine किया जाता है, उन Non-Key Attributes को उनके Determinant के Relation में Specify कर देना चाहिए। इससे Transitive Dependency Remove हो जाती है और इससे Associated Anomalies भी Remove हो जाती हैं, साथ ही हमारा Relation Third Normal Form में आ जाता है। यदि किसी Second Normal Form के Relation में कोई Transitive Dependency ना हो, तो वह Relation Automatically Third Normal Form में आ जाता है।

Boyce-Codd Normal Form

ज्यादातर Relations के लिए Third Normal Form तक Normalized Relation एक अच्छा Design Objective होता है। इस स्थिति के Relations ज्यादातर Anomalies से Free होते हैं।

फिर भी परिस्थितिवश कई बार Third Normal Form के Relations में भी थोडी अलग किस्म की Anomalies होती हैं।

इन Anomalies को **BCNF** व **Forth Normal Form** के Normalization के आधार पर Handle किया जाता है। यदि हमारा Relation Third Normal Form में हो और उसमें कोई Extra Ordinary प्रकार की समस्या ना हो, तो हमारा Relation Automatically BCNF व Fourth Normal Form में होता है। BCNF की प्रक्रिया को समझने के लिए हम एक उदाहरण ले रहे हैं।

मानलो कि Music Store Organization ये तय करता है कि वह अपने Database में एक और Relation Add करेगा, जिसे वह अपने Music Store के Employee के काम करने के समय को Schedule करने के लिए Use करेगा।

हर Employee हर रोज **4-Hours** की एक या दो Shift में काम करेगा और हर Shift में एक Employee को Music Store के किसी एक **Station** (यानी **Stock Room** में Stock को Manage करने के लिए या फिर Desk के सामने **Customer** को Handle करने के लिए) पर काम करेगा, जबकि एक Station पर एक Shift में सिर्फ एक ही Employee काम करेगा। अब Schedule को Handle करने के लिए निम्नानुसार एक Relation Design किया जा सकता हैः

```
Schedule (EmpID, Date, Shift, Station, WorkedShift?)
```

दिए गए Business Rule के हिसाब से एक Employee एक Shift में एक Station पर काम करेगा, इसलिए इस Relation में दो सम्भावित Primary Keys **EmpID + Date + Shift** या **Date + Shift + Station** हो सकती हैं। इस स्थिति में Functional Dependency का Relation निम्नानुसार बनेगाः

```
EmpID + Date + Shift -> Station, WorkedShift?
Date + Shift + Station -> EmpID, WorkedShift?
```

एक बात ध्यान में रखें कि ये Functional Dependency Relation उसी स्थिति में सही हैं, जब हर Station पर हर Shift में केवल एक ही Employee काम करता है। ये Schedule Relation Composite Candidate Keys को Show कर रहा है। क्योंकि दोनों ही Candidate Keys में Date व Shift Common हैं। Boyce-Codd Normal Form को किसी Relation में Exist इसी तरह की Characteristics को Handle करने के लिए बनाया गया है। BCNF Form में होने के लिए किसी भी Relation पर इस नियम का Apply होना जरूरी होता है कि Relation Third Normal Form में हो और Relation के सभी Determinants **Candidate Keys** हों, तो Relation BCNF Form में होता है।

Forth Normal Form

BCNF की तरह ही Forth Normal Form को भी किसी Relation की एक Special Characteristic को Handle करने के लिए Design किया गया है, जो कि बहुत ही कम परिस्थितियों में Generate होती है। इस स्थिति में जो Special Characteristics होती है, उसे सामान्यतया **Multi-Valued Dependency** कहा जाता है। उदाहरण के लिए निम्न Relation को देखिए:

MovieInfo (**Title**, **Star**, **Producer**)

किसी Specify की गई Movie में एक से ज्यादा Stars हो सकते हैं और उसी Movie को एक से ज्यादा Producers ने Produce किया हो सकता है। Same Stars एक से ज्यादा Movie में Appear हो सकते हैं और Producer भी एक से ज्यादा Movies में Involved हो सकता है। इसलिए इस Relation के सभी Columns को Composite Primary Key के रूप में Use करना जरूरी हो जाता है। इस उदाहरण को हम निम्न सारणी में देख सकते हैं:

MovieInfo Table

Title	Star	Producer
Great Film	Lovely Lady	Money Bags
Great Film	Handsome Man	Money Bags
Great Film	Lovely Lady	Helen Pursestrings
Great Film	Handsome Man	Helen Pursestrings
Boring Movie	Lovely Lady	Helen Pursestrings
Boring Movie	Precocious Child	Helen Pursestrings

चूंकि इस Relation में कोई भी Non-Key Attribute नहीं है, इसलिए ये Relation BCNF Normalization Form में है। फिर भी ये Relation निम्न Anomalies Show करता है:

1. हम कम से कम एक Producer को जाने बिना किसी Movie के Stars को Insert नहीं कर सकते हैं।
2. हम कम से कम एक Star को जाने बिना किसी Movie के Producer को Insert नहीं कर सकते हैं।
3. यदि हम किसी Only Producer की Information को Delete करते हैं, तो हम उस Movie के Starts को भी Loose कर देते हैं।
4. यदि हम किसी Movie से Only Star को Delete करते हैं, तो हम उस Movie के Producer की Information को भी Loose कर देते हैं।
5. Movie के हर Star के लिए Producer के नाम का Duplication होता है। इसी तरह से हर Producer के लिए Movie के Star के नाम का Duplication होता है। ये Unnecessary Duplication Modification Anomaly की Situation पैदा करता है।

इस Relation में दो Unrelated Entities हैं। पहला Entity Movie व Stars के बीच की Relationship को Handle करता है और दूसरा Movie व Producer के बीच की Relationship को Handle करता है। Practically देखें तो यही Anomaly का मुख्य कारण है, हालांकि Movie, Star व Producer Entities भी Anomaly में Involved हैं।

फिर भी Theoretically जो Anomalies Create हो रही हैं, वे एक ही Relation में Multivalued Dependency के कारण Create हो रही हैं, जिन्हें Forth Normal Form में Eliminate किया जाता है। Forth Normal Form का नियम ये है कि Relation Boyce-Codd Normal Form में हो और उसमें कोई Multi-Valued Dependency ना हो।

किसी Relation में Multi-Valued Dependency तब Exist होती है, जब किसी Attribute A की हर Value के लिए किसी दूसरे Attribute B की बहुत सारी Values Associated हों और किसी Attribute C की हर Value के लिए Attribute A की बहुत सारी Values Associated हों, जबकि Attribute B व C आपस में Independent हों।

हम जिस उदाहरण को Use कर रहे हैं, उस उदाहरण में इस प्रकार की Dependency है। क्योंकि हर Movie Title के लिए Stars या Actors का एक Group है, जो कि Movie से Associated हैं और हर Movie Title के लिए भी Producers का एक Group है, जिनसे Movie Associated है। फिर भी Actors व Producers एक दूसरे से Independent हैं, क्योंकि इनके बीच कोई Direct Connection नहीं है। इस Multivalued Dependency को निम्नानुसार Represent किया जा सकता है:

```
Title ->> Star
Title ->> Producer
```

और इसे इस तरह पढा जाता है कि:

"**Title** Multi-Determines **Star** and Title Multi-Determines **Producer**."

Functional Dependency एक Multi-Valued Dependency का एक Special प्रकार है, जहां सिर्फ एक मान को Determine किया जाता है, ना कि मानों के एक Group को। Multi-Valued Dependency को Eliminate करने व इस Relation को Forth Normal Form में लाने के लिए, हमें Relation को Split करना होता है और Relation की Dependency के हर हिस्से को निम्नानुसार उसके स्वयं के Relation में Place करना होता है

```
MovieStars (Title, Star)
MovieProducers (Title, Producer)
```

इस Design में हम स्वतंत्र रूप से **Stars** व **Producers** को बिना एक दूसरे को प्रभावित किए हुए, **Insert** व **Remove** कर सकते हैं। Stars व Producers का नाम भी हर Movie के लिए केवल एक ही बार Appear होता है, जिनसे वे Connected होते हैं।

किसी Database के Relations को Normalize करने पर हर Entity अपने स्वयं के अलग Relation द्वारा Represent होता है और Normalization हमें ये सुविधा देता है कि हम बिना किसी दूसरे Entity को Directly Disturb करते हुए, विभिन्न Relations में Data को Insert कर सकते हैं, Delete कर सकते हैं, Modify कर सकते हैं। हालांकि Normalization की भी अपनी कुछ कमियां हैं।

हम Relations को इसलिए Split करते हैं, ताकि Relationships को Primary व Foreign Keys के Matching द्वारा Represent किया जा सके। हम जब भी DBMS से किसी Query द्वारा एक से अधिक Tables के Data को प्राप्त करना चाहते हैं, तब हम DBMS को विभिन्न Relations के बीच Matching Operation को Perform करने के लिए बाध्य करते हैं।

उदाहरण के लिए किसी Normalized Database में हम किसी Order के Data को एक Relation में Store करते हैं, Customer के Data को दूसरे Relation में Store करते हैं और Order Lines के Data को तीसरे Relation में Store करते हैं। जब हम Query करते हैं, तब ये Query Operation इन तीनों ही Relations से Required Data को प्राप्त करके एक Single Table के रूप में Prepare करता है। ताकि किसी **Invoice** के लिए Output Generate किया जा सके। विभिन्न Tables के Data को Combined Form में एक Table के रूप में दिखाने के लिए DBMS एक विशेष Process का प्रयोग करता है। इस प्रक्रिया को **Join** कहा जाता है।

Theory के रूप में **Join Operation** दो Relations के बीच Matching Values के आधार पर **Records** को या **Rows** को Search करता है और जितनी बार भी उसे Match प्राप्त होता है, वह Resultant Table में एक नया **Record** या **Row** Create कर देता है।

हालांकि Join Operation को Perform करके एक से ज्यादा Relations से Data को Manipulate किया जाना एक अच्छी प्रक्रिया है। लेकिन जब Join Operations से बहुत ज्यादा Records Access होते हैं, तब DBMS की Performance यानी Data Manipulation की Speed कम हो जाती है।

ये जानने के लिए कि Join Operation से क्या हो सकता है, हमें Join Operation के Algebra को समझना होगा। Relational Algebra, Operations का एक ऐसा समूह है, जिसका प्रयोग किसी Relation से Data को Manipulate व Extract करने के लिए किया जाता है। हर Operation दो Tables पर, एक Single Manipulation Perform करता है। किसी Query को Complete करने के लिए DBMS, Relational Algebra Operations का पूरा एक Sequence Use करता है।

Relational Algebra एक तरह से Procedural होता है, जबकि **SQL,** Relational Calculus पर आधारित होता है। **SQL** में हमें केवल ये बताना होता है कि एक Database Relation से हमें क्या Data चाहिए। जबकि हमें ये बताने की जरूरत नहीं होती है, कि हमें Database Relation से वह Data कैसे चाहिए। एक Single SQL Retrieval Command, DBMS को एक या सभी Relational Algebra Operations को Perform करने के लिए प्रेरित कर सकता है।

Equi – Join

इसके सबसे Common रूप में एक Join Operation उस समय नए Records या Rows Create करता है, जब दो Source Tables के Data आपस में Match होते हैं। क्योंकि हम Rows को Equal Values के लिए खोज रहे हैं, इसलिए इस तरह की Join को Equi-Join या Natural aEqui-Join कहा जाता है। उदाहरण के लिए अगली दो Tables को देखिएः

Customers Table

CustID	FName	LName
001	Jane	Doe
002	John	Doe
003	Jane	Smith
004	John	Smith
005	Jane	Jones
006	John	Jones

Orders Table

OrderID	CustID	OrderDate	OrderTotal
001	002	10/10/99	250.65
002	002	2/21/O0	125.89
003	003	11/15/99	1567.99
004	004	11/22/99	180.92
005	004	12/15/99	565.O0
006	006	10/8/99	25.O0
007	006	11/12/99	85.O0
008	006	12/29/99	109.12

ध्यान दें कि CustID Column Customer के Relation का Primary Key है और यही CustID Column Foreign Key की तरह Orders Table में भी है। इसलिए Orders Table का CustID उन Customers से Belong करता है, जिन्होंने Order Place किया है।

मानलो कि हम उन Customers के नाम जानना चाहते हैं, जिन्होंने Order Place किया है। ये जानकारी प्राप्त करने के लिए हमें दो Tables को CustID Column के आधार पर Combined Rows Create करके Join करना होगा।

यदि हम Database के शब्दों में कहें तो हम कह सकते हैं कि हम CustID के आधार पर दो Tables को Join कर रहे हैं। Join करने पर हमें प्राप्त होने वाली Resultant Table निम्नानुसार होती है:

Result Table

CustID	FName	LName	OrderID	OrderDate	OrderTotal
002	John	Doe	001	10/10/99	250.65
002	John	Doe	002	2/21/00	125.89
003	Jane	Smith	003	11/15/99	1597.99
004	John	Smith	004	11/22/99	180,92
004	John	Smith	005	12/15/99	565.00
006	John	Jones	006	10/8/99	25,00
006	John	Jones	007	11/12/99	85.00
006	John	Jones	008	12/29/99	109.12

Equi-Join को हम एक Table पर भी पूरी तरह से Apply कर सकते हैं। जब हम Equi-Join करते हैं, तब Join एक Source के हर Row को दूसरी Table के हर Row से Compare करता है। First Source Table की हर Row के लिए ये Second Source Table के Columns में Matching Data खोजता है और जैसे ही कोई Matching Row मिल जाता है, ये Result Table में एक नया Row Place कर देता है।

मानलो कि हम First Source के रूप में Customers Table को Use कर रहे हैं और Second Source के रूप में Orders Table को, तो Result Table निम्नानुसार Produce होती है:

1. सबसे पहले CustID 001 के लिए Orders Search किया जाता है। चूंकि Orders Table में इस ID के लिए कोई Matching Row नहीं है, इसलिए Equi-Join Result Table में कोई Row Place नहीं करता है।
2. इसके बाद CustID 002 के लिए Orders को Search किया जाता है। चूंकि इस ID की दो Matching Rows Orders Table में हैं, इसलिए Equi-Join Result Table में दो Rows Place करता है और दोनों Rows में दो बार Same Customer Information को Store करके Order की Information को Store करता है।

3. इसके बाद CustID 003 के लिए Orders को Search किया जाता है। इस ID से Related एक Order है, इसलिए Equi-Join एक और नया Row Result Table में Place कर देता है।
4. फिर CustID 004 के लिए Orders को Search किया जाता है। इस ID से Related दो Matching Orders हैं, इसलिए Result Table में दो नए Rows को Add किया जाता है।
5. फिर CustID 005 के लिए Orders को Search किया जाता है। इस ID से Related कोई Matching Orders नहीं हैं, इसलिए Result Table में कोई नया Row Add नहीं किया जाता है।
6. फिर CustID 006 के लिए Orders को Search किया जाता है। इस ID से Related तीन Matching Orders हैं, इसलिए Result Table में तीन नई Rows को Add किया जाता है।

ध्यान दें कि यदि CustID दोनों Tables में Appear ना हो, तो कोई भी Row Result Table में Place नहीं होता है। Join के इस व्यवहार को **Inner Join** Group में रखा जाता है। यानी इस तरह की Joining को Inner Join कहा जाता है।

एक Join Operation को दो अन्य Operations जिन्हें **Product** व **Restrict** Operation कहा जाता है, के रूप में भी Implement किया जा सकता है। इस तरह के Operations में बहुत ज्यादा Data के साथ Manipulation होता है, इसलिए यदि DBMS इस तरह के Operations Perform करता है, तो Database बहुत ही धीमें काम करता है और Query की Performance बहुत ही कम हो जाती है।

Restrict Operation किसी Table के Matching Rows को छोडकर शेष Rows को Retrieve कर लेता है। जबकि Product Operation दोनों Tables के हर Row की Cartesian Product के रूप में जितने सम्भव हों, उतने Pair Create करता है।

उदाहरण के लिए यदि **Customer** व **Orders** Table पर इस Operation को Perform किया जाए, तो Customer Table में **6 Rows** हैं जबकि Orders Table में **8 Rows** है अतः परिणामस्वरूप Result Table में कुल **48 Rows** Create होंगे। इस Operation में **CustID** Column दो बार Appear होता है, क्योंकि ये Column दोनों Tables में Exist है।

DBMS

DATABASE STRUCTURE AND PERFORMANCE TUNING

DBMS – Database Structure and Performance Tuning

Database Design करने के अलावा DBA को एक काम और करना होता है और वह काम होता है Database Performance की Tuning करने का। Database की Performance को ठीक तरह से Tune ना करने पर Database के काम करने की Speed काफी कम हो जाती है। Database की Speed को Tune करने के लिए हमें Database के Design में भी Modification करना पडता है।

लगभग हमेंशा एक DBMS ही User के Commands के आधार पर Database में Data को Store करने या Database से Data को Retrieve करने का काम करता है। जिस तरीके का प्रयोग करके एक DBMS Software किसी User Request को पूरा करने के सभी Data Manipulation Operations को DBMS का **Query Optimizer** ही Perform करता है। Query Optimizer, DBMS Software का एक ऐसा हिस्सा होता है, जो किसी Query को Perform करने के लिए Relational Algebra Operation के सबसे Efficient Sequence को तय करने का काम करता है।

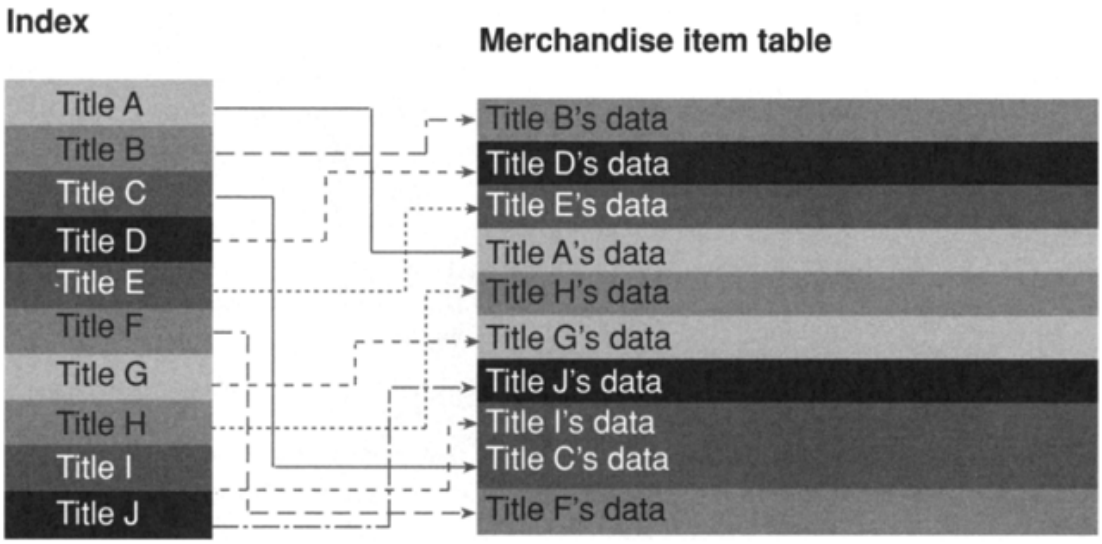
हालांकि Query Optimizer के काम करने के तरीके को एक Database Designer किसी भी तरह से Directly Handle नहीं कर सकता है, लेकिन Database के Design में कुछ व्यवस्थाएं करके हम Database की Performance को कुछ हद तक Increase कर सकते हैं।

Indexing

Indexing एक ऐसा तरीका होता है, जो किसी Column या Composite Columns के Data को Access करने का Fast तरीका प्रदान करता है। एक Database Application Use करने वाला User जितने भी Records किसी Table में Add करता जाता है, वे सभी Records Table के अन्त में Random Order में जुडते जाते हैं।

जैसे–जैसे किसी Table के Records की संख्या बढती जाती है, वैसे–वैसे Table से Sequential Search द्वारा किसी Record के Search होने की Process धीमी होती जाती है। बिना किसी एक अच्छे तरीके को Use किए हुए DBMS किसी Value को Search करने के लिए हमेंशा Sequential तरीके का प्रयोग करता है, जिसमें DBMS किसी Column को Top से Bottom की तरफ Scan करता है।

इसलिए Table में Records की संख्या जितनी ज्यादा होती है, Sequential Search की Speed उतनी ही कम होती जाती है। Indexing के Conceptual Operation Diagram को निम्न चित्र में दर्शाया गया है:



इस चित्र में हम Orders Table के विभिन्न Records का Relation एक Index Table के साथ देख रहे हैं। ये Index हमेंशा Sorted Form में रहता है, इसलिए इस पर विभिन्न प्रकार के अन्य Operations Perform करके एक Database किसी Record को ज्यादा Fastly Search कर सकता है। Index में हर Record के Keys की एक Ordered List होती है, जिसके साथ Order Table का हर Record Associated रहता है। हालांकि Order Table के सभी Records Random Order में हैं, लेकिन Index Table में सभी Records Sorted Order में होने की वजह से Records को Fastly Search किया जा सकता है।

जब एक बार हम Index Create कर देते हैं, उसके बाद जब भी जरूरत होती है, तो DBMS का Query Optimizer इस Index का प्रयोग करके ही किसी Record को Search करता है। हमें इस Index को दुबारा Access करने की तब तक कोई जरूरत नहीं होती है, जब तक कि हम इस Index को Delete करना नहीं चाहते हैं।

जब हम किसी Table में कोई Primary Key Create करते हैं, तो DBMS इस Primary Key या Composite Key के Columns के आधार स्वयं ही एक Index Create कर लेता है। जब भी हम किसी Table में कोई नया Record Insert करते हैं, उस Record के Primary Key के मान को Uniqueness के लिए DBMS द्वारा Check किया जाता है। इस Uniqueness के लिए Directly Base Table के Primary Key को Check करने के बजाय DBMS उस Index को Check करता है और चूंकि Index एक Ordered Form में होता है, इसलिए ये Verification काफी तेजी से हो जाता है।

ऐसा जरूरी नहीं होता है कि DBMS हमेंशा हमारे Primary Key के आधार पर ही Index Create करेगा। वास्तव में हम स्वयं भी हमारी Table के किसी भी **Column** या **Group Of Columns** के आधार पर Index Create कर सकते हैं। लेकिन Indexing के साथ कुछ **Trade-Offs** भी हैं, जो निम्नानुसार हैं:

1. Indexes Database में Extra Space लेते हैं। चूंकि आज Disk Space ज्यादा महंगा नहीं है, इसलिए Indexes के सम्बंध में आज ये कोई बडी समस्या नहीं है।
2. जब हम किसी Indexed Column के Record में किसी Data को Insert या Modify करते हैं, या उस Record को Delete करते हैं, तो DBMS Base Table के साथ ही उस Index को भी Update करता है। इस प्रक्रिया के कारण Data Modification की प्रक्रिया धीमी हो जाती है, विशेष रूप से तब जब Table में Records की संख्या काफी ज्यादा होती है।
3. फिर भी Indexes Data के Access को निश्चित रूप से Increase करते हैं।

सामान्यतया Update Speed व Retrieval Speed के बीच Trade-Off होता है। Indexing के लिए एक उचित नियम ये है कि Indexing के लिए उन Columns को Choose करना चाहिए, जो SQL Query में ज्यादा Use होते हैं और Indexing सामान्यतया **Foreign Key Columns** की करनी चाहिए। यदि किसी Indexing को Apply करने पर हमें लगता है कि Update Speed ज्यादा प्रभावित हो रही है, तो हमने जिन Indexes को Create किया है, उनमें से कुछ को या सभी को जरूरत के आधार पर Delete कर सकते हैं।

Clustering

Disk पर Data को Write करना या Disk से Data को Read करना DBMS का सबसे Slowest काम होता है। यदि हम Data के Disk पर Store होने व Disk से Data के Retrieve होने की संख्या को कम कर सकें, तो हम DBMS की Performance को बढा सकते हैं।

Computer में सभी Records Disk Page के रूप में Store होते हैं। जब भी हम किसी Record को प्राप्त करने की Request करते हैं, Database उस Record के पूरे एक Page को Retrieve करता है, जिसमें वह Record होता है। Page की Size अलग–अलग Operating Systems के आधार पर बदलती रहती है। Page की Size 512 Bytes से लेकर 4 KBytes तक होती है।

हमें Disk से भले एक ही Record की जरूरत क्यों ना हो, हमेंशा Disk से सम्बंधित Record का पूरा एक Page ही Access होता है। इसलिए यदि हम उन Data को Access कर रहे हैं, जो समान Disk Page पर Stored हैं या जो नजदीकी Page में Stored हैं, तो हम Data Access की Speed को बढा सकते हैं। इस Process को **Clustering** कहते हैं और इसकी सुविधा Oracle जैसे DBMS में उपलब्ध है।

Cluster को Primary व Foreign Keys के Matching से बनने वाले Records को Hold करने के लिए Design किया जाता है। Cluster को Define करने के लिए हमें उन Tables के Column या Columns के Group को Specify करना होता है, जिनके आधार पर DBMS Cluster Create करता है और उन Tables को Cluster में Include करता है।

फिर जिन Column या Composite Columns के आधार पर Clusters Create किया गया है, उन Column या Composite Columns के Same Values को Share करने वाले Records को Disk पर Physically Store किया जाता है। इन Records को जितना सम्भव होता है उतना नजदीक पर Store किया जाता है।

परिणामस्वरूप किसी Table के विभिन्न Records कई Disk Pages के रूप में बिखरे हुए रहते हैं, लेकिन Matching Primary Keys व Foreign Keys के Records अक्सर Same Page पर ही Store होते हैं। Clustering से सामान्यतया Join Performance की Speed बढ जाती है। फिर भी Indexes की तरह ही Clusters Create करने से सम्बंधित भी कुछ Trade-Offs हैं, जो निम्नानुसार हैं:

1. चूंकि Clustering Data के किसी File में Physically Store होने से सम्बंधित होता है, इसलिए एक Table को केवल एक Column या एक Composite Column के आधार पर ही Clustered किया जा सकता है।
2. जब पूरी Table के Records को Scan करने की जरूरत होती है, तब Clustering की वजह से Scanning की Speed कम हो जाती है, क्योंकि Clustering के कारण एक ही Table के विभिन्न Records Disk पर विभिन्न Disk Pages में Spread होकर Store होते हैं।
3. Clustering से Data Insertion की Speed में भी कमी आती है।
4. Cluster जिस Column या Composite Column पर आधारित होता है, उन Columns का Modification करने से Speed कम हो जाती है।

Partitioning

Clustering की Reverse प्रक्रिया को **Partitioning** कहा जाता है। ये किसी बडी Table को कई छोटी Tables में Divide कर देता है, ताकि DBMS बहुत सारे Data को एक साथ Retrieve ना कर सके।

उदाहरण के लिए यदि हम Music Store Application के Database को लें, तो जैसे–जैसे Customers के Orders की संख्या बढती जाती है, विशेष रूप से **Order Lines** Table के Records की संख्या काफी बढ जाती है। यदि इन दोनों Tables में Records की संख्या काफी ज्यादा हो जाए, तो इनसे किसी Record को Retrieve करने की Speed काफी कम हो जाएगी।

किसी Table का Horizontally व Vertically दो तरीकों से Partition किया जा सकता है। **Horizontal Partitioning** में एक Table के विभिन्न Rows या Records को Identical Structure में दो या दो से अधिक Tables में Split कर दिया जाता है।

जबकि **Vertical Partitioning** में किसी Table के Columns को आपस में एक Primary Key द्वारा Linked रखते हुए Split कर दिया जाता है। दोनों ही Partition Process के अपने कुछ फायदे व कुछ नुकसान हैं।

Horizontal Partition में एक Table को Records के आधार पर दो या दो से अधिक Tables में Split किया जाता है, जबकि दोनों ही Tables का Structure समान रखा जाता है। Music Store Database में हम इस तकनीक को Use कर सकते हैं।

उदाहरण के लिए Orders व Line Items Table को यदि Horizontal Partitioning के आधार पर एक से ज्यादा Tables में Divide करना हो, तो हम इस काम को निम्नानुसार कर सकते हैं:

```
OpenOrders (OrderID, CustID, OrderDate)
OpenOrderLines (OrderID, ItemID, Quantity, Shipped?)
FilledOrders (OrderID, CustID, OrderDate)
FilledOrdersLines (OrderID, ItemID, Quantity, Shipped?)
```

जब भी OpenOrders Table के सभी Items को Ship कर दिया जाता है, एक Application Program OpenOrders Table व OpenOrderLines Table के सभी Records को Delete कर देता है और इन Records को FilledOrders व FilledOrdersLines Table में Fill कर देता है।

इस प्रक्रिया के कारण OpenOrders व OpenOrderLines Table दोनों में ही Records की संख्या कम ही रहती है जिससे Data के Modification व Retrieval की Performance बढ जाती है। हालांकि FilledOrders व FilledOrdersLines Table से Data के Retrieval की Speed काफी धीमी होती है, लेकिन Music Store इन Tables को बहुत कम बार Access करता है।

इस तरीके के साथ तब परेशानी आती है जब Music Store को Orders Table या OrderLines Tables के सभी Records को एक साथ Access करने की जरूरत पडती है। इस तरीके को Use करने पर जब हम किसी Query में इन दोनों Tables के Data को Access करना चाहते हैं, तब हमें **UNION** Operator का प्रयोग करते हुए दो Queries को Mix करके Data को Access करना पडता है। यदि हम जो Application Create कर रहे हैं, उसमें दोनों Tables को बहुत कम बार एक साथ Access करने की जरूरत पडती है, तो हम इस तरीके को Performance बढाने के लिए Use कर सकते हैं।

Horizontal Partitioning से हमारे Database की Performance बढेगी या नहीं, इस बात का पता लगाने का केवल एक ही तरीका है, कि हम ये जानने की कोशिश करें कि हमारा Application इस तरह के Data को किस तरह से Access करने वाला है। यदि कुछ **Records** का एक ऐसा

Group हो जिसे बार–बार Access किए जाने की जरूरत पडती हो, तो हम इस तरह की Partitioning को अपने Database पर Apply कर सकते हैं।

Vertical Partitioning में एक ही Table के विभिन्न Columns को एक से ज्यादा Tables में Divide कर लिया जाता है और दोनों ही Tables की Primary Key को समान रखा जाता है। ऐसा करने पर सभी Tables आपस में One To One की Relationship से Lined रहती हैं।

उदाहरण के लिए यदि Music Store के Database में Titles व Prices की Information को काफी ज्यादा बार Use करने की जरूरत पडती है, तो हम Vertical Partitioning को Table पर Apply करके उसे निम्नानुसार दो भागों में बांट सकते हैं:

```
ItemTitles (ItemID, Title, Price)
ItemDetails (ItemID, Distributor, ReleaseDate, . . . )
```

इस Design का फायदा ये है कि ItemTitles Table के Records Physically काफी Close होते हैं। छोटी Tables कम Disk Pages में Store होते हैं, इसलिए इस प्रकार के Tables की Performance काफी अच्छी होती है। जब दोनों ही Tables के Data की जरूरत होती है, तब दोनों ही Tables को ItemID के आधार पर Join किया जाता है। अन्य Join Operation की तरह ही इस Join Operation की Speed भी कम होती है।

Last but not Least. There is more…

Computer System व विभिन्न Programming Languages का विकास करने का मुख्य कारण Business Solusions Develop करना ही रहा है। इसीलिए सबसे पहले Develop किया गया Oracle नाम का DBMS Software वर्तमान समय में भी उतना ही उपयोगी है, जितना इसके Development के समय उपयोगी था।

यानी आज भी यदि कोई Application Software सबसे ज्यादा Develop किया जाता है, तो वह Business Solution Software ही होता है, जो कि पूरी तरह से Database Application होता है, जिसमें किसी Company या Business की किसी Specific Type की Problem को Manage व Solve किया जाता है।

इस पुस्तक में हमने मूल रूप से एक Database Application Software Develop करने से पहले की जाने वाली तैयारी के बारे में ही विस्तार से Discuss किया है ताकि आप समझ सकें कि एक Application Software को Develop करने से पहले उसके लिए Develop किया जाने वाला Data किस प्रकार का हो, ताकि भविष्य में किसी प्रकार की Database Related Anomaly पैदा न हो।

ये पुस्तक मूल रूप से उन लोगों के लिए काफी उपयोगी है जो Database Administrator यानी DBA Level के Profesisonals बनना चाहते हैं, क्योंकि वर्तमान समय में बनने वाले लगभग हर Database आधारित Application के लिए किसी न किसी प्रकार का Database तो Design करना ही पडता है और इसी Database में ही Develop किए जाने वाले Application का सारा Data Stored रहता है।

इस पुस्तक में हमने Database Design करने से सम्बंधित जिन Concepts को Discuss किया है, वे सभी Concepts किसी भी Relational Database System Software जैसे कि **MS-SQL Server, MSAccess, Oracle, Sybase, MySQL, DB2** आदि पर समान रूप से Apply होते हैं। इसलिए यदि आप इन Database Related Concepts को किसी किसी एक Software के लिए सीख लेते हैं, तो किसी भी अन्य Software के लिए इन्हें सीखना काफी आसाना हो जाता है।

उम्मीद है, इस पुस्तक ने आपके Professional Database Application Develop करने से सम्बंधित ज्ञान को जरूर बढाया होगा और अब आप किसी भी Problem को Solve करते समय ज्यादा आसानी से इस बात का निर्णय लेने में सक्षम हैं, कि किस प्रकार के Data को Store करने के लिए किस प्रकार का Database व Table Create करना होगा तथा उस Database की विभिन्नन Tables के Data के बीच किस प्रकार की Data Relationship होगी।